

```

;;; demo_zos.asm
;;;
;;; demonstration (and, frankly, bring-up) app for zOS
;;; to build: gpasm -D GPASM demo_zos.asm
;;;
;;; after starting job #1 as a console output buffer (zOS_CON() in zosmacro.inc)
;;; to demonstrate privileged mode (able to kill or otherwise tweak other tasks)
;;;
;;; it starts a splash() job #2 to copy a packed ascii greeting into the buffer
;;; (using the SWI line zOS_SI3) character by character, also privileged so that
;;; it can un-wait the two unprivileged tasks (to guarantee they don't overwrite
;;; the potential long greeting)
;;;
;;; two final processes (should end up numbered jobs 3 and 4) run in re-entrant
;;; function splitjob() printing their own job numbers to the console
;;;
;;; since only 4 of 5 possible task slots are used in this demo reducing the max
;;; allowed value by 1 will make scheduler run faster:
zOS_NUM equ 4

processor 16f1719
include p16f1719.inc

__CONFIG __CONFIG1,_FOSC_INTOSC & _WDTE_OFF & _PWRTE_OFF & _CP_OFF & _BOREN_
ON & _CLKOUTEN_ON & _IESO_ON & _FCMEN_ON
__CONFIG __CONFIG2,_WRT_OFF & _PPS1WAY_OFF & _ZCDDIS_ON & _PLLEN_ON & _STVRE
N_ON & _BORV_LO & _LPBOR_OFF & _LVP_ON

;;; uncomment to reduce zOS footprint by 100 words (at cost of zOS_FRK/EXE/FND):
zOS_MIN equ 1

include zos.inc
include zosmacro.inc

OUTCHAR equ zOS_SI3

;;; uncomment to pre-load stack positions with indices (for debugging ZOS_ROL):
; zOS_DBG

pagesel main
goto main

greet
da "Demo application for zOS"

crlf
da "\r\n",0

put_str
zOS_STR OUTCHAR
return ;void put_str(const char*) { zOS_STR(OUTCHAR); }
SPLVAR equ 0x20
splash
movf zOS_ME ;void splash(void) {
zOS_ARG 0 ; // ceding processor to let both spitjob()s run
zOS_SWI zOS_YLD ; zOS_ARG(0, bsr);
movf zOS_ME ; zOS_SWI(zOS_YLD);
zOS_ARG 0 ; zOS_ARG(0, bsr);
zOS_SWI zOS_YLD ; zOS_SWI(zOS_YLD);
zOS_ADR greet,zOS_FL A ;
pagesel put_str ; zOS_ADR(fsr0="Demo application for zOS\r\n");
call put_str ; put_str(fsr0);
movlw zOS_NUM+1 ; uint8_t splvar = zOS_NUM + 1;
movwf SPLVAR ; while (--splvar) {

splalp
movlw low spitjob ; zOS_ARG(0, spitjob & 0x00ff);
zOS_ARG 0
movlw high spitjob ; zOS_ARG(1, spitjob >> 8);
zOS_ARG 1
decf SPLVAR,w ; zOS_ARG(2, splvar); // max job# to find
btfsc STATUS,Z ; splvar = zOS_SWI(zOS_FND);

```

```

bra spldone ; if (splvar)
zOS_ARG 2
zOS_SWI zOS_FND
movwf SPLVAR ; zOS_UNW(splvar); // un-wait found spitjob()s
movf SPLVAR,f ; else
btfsc STATUS,Z ; break; // until none found at all
bra spldone ; }
zOS_UNW SPLVAR
bra splalp ; zOS_ARG(0, bsr);

spldone
movf zOS_ME ; zOS_SWI(zOS_END); // unschedule self
zOS_ARG 0 ;}
zOS_SWI zOS_END

spitjob
zOS_SWI zOS_WAI ;void spitjob(void) {

reprint
movf zOS_ME ; zOS_SWI(zOS_SLP); // splash() wakes when done
andlw 1 ; do {
brw ; w = zOS_ME(); // shouldn't get clobbered below
bra asxbyte ; switch (w & 1) {
bra asascii ; case 0:

asxbyte
clrw ; zOS_ARG(0, 0);
zOS_ARG 0
movf zOS_ME ; zOS_ARG(1, w); // print as numeric "02"/"03"
zOS_ARG 1
bra print ; break;

asascii
movlw '0' ; case 1:
addwf zOS_ME ; zOS_ARG(0, w); // print as character '2'/'3'
zOS_ARG 0 ; }

print
zOS_SWI OUTCHAR ; zOS_SWI(OUTCHAR);
zOS_ADR crlf,zOS_FL A ; zOS_ADR(fsr0 = "\r\n");
pagesel put_str
call put_str ; put_str(fsr0);

#if 1
spit_i equ 0x20
spit_j equ 0x21
loop
incfsz spit_j,f ; for (int i = 0; i & 0xff; i++)
bra loop ; for (int j = 0; j & 0xff; j++)
incfsz spit_i,f ; ;
bra loop ; } while (1);

#endif
bra reprint ;}

;;; while SWI handlers normally know what line the interrupts will come in on,
;;; for flexibility of incorporation into any application this choice is not
;;; hardwired into zosmacro.inc library and any available line may be chosen:

main
banksel OSCCON ; {
movlw 0x70 ; // SCS FOSC; SPLLEN disabled; IRCF 8MHz_HF;
movwf OSCCON ; OSCCON = 0x70;
movlw 0x80 ; // SOSCR enabled;
movwf OSCSTAT ; OSCSTAT = 0x80;
movlw 0x00 ; // TUN 0;
movwf OSCTUNE ; OSCTUNE = 0x00;
; // Wait for PLL to stabilize
btfss OSCSTAT,PLLR ; while(PLLR == 0)
bra $-1 ; ;

banksel ANSELA
movlw 0xaf ;
movwf ANSELA ; ANSELA = 0xaf; // allow heartbeat GPIO, CLKOUT
movlw 0x3c ;
movwf ANSELC ; ANSELC = 0x3c; // allow serial port

```

```

banksel OPTION_REG
bcf    OPTION_REG,PSA    ; OPTION_REG &= ~(1<<PSA); // max timer0 prescale
bcf    OPTION_REG,T0CS   ; OPTION_REG &= ~(1<<TMR0CS); // off Fosc not pin

banksel TRISC
bcf    TRISA,RA4         ; TRISA &= ~(1<<RA4); // allow heartbeat output
; bcf    TRISA,RA6         ; TRISA &= ~(1<<RA6); // allow clock output
movlw  0x7f
movwf  TRISC

banksel PPSLOCK
movlw  0x55
movwf  PPSLOCK
movlw  0xaa
movwf  PPSLOCK
bcf    PPSLOCK,PPSLOCKED
movlw  0x16
movwf  RXPPS

banksel RC7PPS
movlw  0x14
movwf  RC7PPS
movlw  0x55
movwf  PPSLOCK
movlw  0xaa
movwf  PPSLOCK
bsf    PPSLOCK,PPSLOCKED

zos_MAN 0,.32000000/.9600,PIR1,LATA,RA4
movlw  OUTCHAR           ;void main(void) {
movwi  0[FSR0]           ;
; zos_CON 0,.32000000/.9600,PIR1,LATA,RA4
; zos_ARG 3               ; zos_CON(*UART*/1,20MHz/9600bps,PIR1,PORTB,5);
; zos_LAU WREG            ; zos_ARG(3,OUTCHAR/*only 1 SWI*/); zos_LAU(&w);

; zos_INT 0,0             ; zos_INT(0,0); //no interrupt handler for splash
; zos_ADR splash,zos_PRB  ; zos_ADR(fsr0 = splash&~zos_PRV); // privileged
; zos_LAU WREG            ; zos_LAU(&w);

; zos_INT 0,0             ; zos_INT(0,0); //no interrupt handler either
; zos_ADR spitjob,zos_UNP ; zos_ADR(fsr0 = spitjob&~zos_PRV); //unprivilege
; zos_LAU WREG            ; zos_LAU(&w);
; zos_LAU WREG            ; zos_LAU(&w); // launch two copies

zos_RUN INTCON,INTCON    ; zos_RUN(*T0IE in*/INTCON, /*T0IF in*/INTCON);
end                      ;}

```

```

;;; zos.inc
;;; a lightweight, small-footprint, preemptively multitasking RTOS for Microchip
;;; Technology's entire enhanced midrange 8-bit PIC microcontroller family:
;;;
;;; jobs (up to 5) are never allowed to manipulate the BSR directly, as that is
;;; the prerogative of zOS (it being used as the current job #) and the bank may
;;; never end up greater than zOS_NUM in user space with interrupts enabled!!!

;;; memory footprint:
;;; ~613 14-bit words for base RTOS i.e. main() starts at 0x0263
;;; ~511 words if zOS_MIN is defined to omit FRK/EXE/FND (thus SWI#4~7=zOS_YLD)
;;;
;;; SRAM footprint:
;;; 86 bank-0 bytes claimed by RTOS, 30 bytes of stack scratch space relocatable
;;;
;;; available bytes      possible jobs with      local bytes/job (+any heap, besides
;;; on PIC device        80 bytes RAM each        2 global bytes) if zOS_NUM set to 5
;;; =====
;;;      128              0                      0 (+2)
;;;      256              1                      0 (+130)
;;;      384              3                      0 (+258)
;;;      512              4                      0 (+386)
;;;      768              5                      80 (+242)
;;;     1,024             5                      80 (+498)
;;;     2,048             5                      80 (+1522)
;;;     4,096             5                      80 (+3570)

;;; you may redefine a constant zOS_NUM with the maximum job number (<6,
;;; as determined by where the general purpose register memory stops, as
;;; the guaranteed 2 bytes global memory isn't sufficient for most jobs)
#ifdef zOS_NUM
#else
zOS_NUM set      5
#endif

;;; you may redefine the location of the scratch space for restoring the stack
;;; after each context switch (by default it is 0x20 in bank zOS_NUM+1, but can
;;; be pulled in on small devices into unused local storage, or pushed out if necc
#ifdef zOS_STK
#else
zOS_STK set      (((zOS_NUM+1)<<7)|0x20)
#endif
#ifdef zOS_FRE
#else
zOS_FRE set      (0x2000+((zOS_NUM+1)*0x50)+(0x001e))
#endif

;;; software interrupt infrastructure zOS is based on (even with interrupts off)

;;; 5 user-definable software interrupt lines:
zOS_SB7 equ      7
zOS_SI7 equ      (1<<zOS_SB7)
zOS_SB6 equ      6
zOS_SI6 equ      (1<<zOS_SB6)
zOS_SB5 equ      5
zOS_SI5 equ      (1<<zOS_SB5)
zOS_SB4 equ      4
zOS_SI4 equ      (1<<zOS_SB4)
zOS_SB3 equ      3
zOS_SI3 equ      (1<<zOS_SB3)

;;; 7 system software interrupts for job management:
zOS_FND equ      0x07      ; find a running job <=AR2 by its handle AR1:AR0
zOS_EXE equ      0x06      ; replace this job with a new job (unpriv'ed)
zOS_FRK equ      0x05      ; copy a running job into a new job
zOS_YLD equ      0x04      ; (in)voluntarily cede processor before next irq
zOS_RST equ      0x03      ; restart job at its start address (vs. END+NEW)
zOS_END equ      0x02      ; job killed, slot# available for NEW
zOS_SLP equ      0x01      ; indicate job waiting on its ISR, so don't run

```

```

zOS_NEW equ      0x00      ; create a job (FSR0==addr,AR1:0==isr,AR3:2==IM)

;;; global memory space for 2 scratch registers plus message-passing mailboxes
zOS_JOB equ      0x70      ; next job to run (0 if unknown)
zOS_MSK equ      0x71      ; masked-off software interrupt for ISR to handle
zOS_J1L equ      0x72      ; (repurposeable as scratch after zOS_RFS call)
zOS_J1H equ      0x73
zOS_J2L equ      0x74
zOS_J2H equ      0x75
zOS_J3L equ      0x76
zOS_J3H equ      0x77
zOS_J4L equ      0x78
zOS_J4H equ      0x79
zOS_J5L equ      0x7a
zOS_J5H equ      0x7b
; must disable interrupts e.g. with zOS_ARG(0) before writing SWI args:
zOS_AR0 equ      0x7c
zOS_AR1 equ      0x7d
zOS_AR2 equ      0x7e
zOS_AR3 equ      0x7f

;;; job/shadow register offsets from zOS_J0M, zOS_J1M,...
zOS_HDL equ      0x00      ; handle, the start address of the job
zOS_HDH equ      0x01      ;
zOS_PRB equ      7        ; MSB of HDH indicates privilege(manage others)
zOS_RAM equ      0        ;
zOS_FLA equ      1        ;
zOS_UNP equ      0        ;
zOS_PCL equ      0x02      ; address to resume execution
zOS_PCH equ      0x03      ; "impossible" PCH 0x00==not runnable
zOS_WAI equ      7        ; MSB of PCH indicates sleeping (wait for int)
zOS_SST equ      0x04      ; shadow STATUS
zOS_SWR equ      0x05      ; shadow WREG
zOS_SSP equ      0x06      ; STKPTR to be restored (BSR implied by base)
zOS_SPH equ      0x07      ; PCLATH to be restored
zOS_SF0 equ      0x08      ; shadow FSR0
zOS_SF1 equ      0x0a      ; shadow FSR1
zOS_ISR equ      0x0c      ; interrupt service routine address for the job
zOS_ISH equ      0x0d      ; interrupt service routine address for the job
zOS_HIM equ      0x0e      ; mask for hardware interrupts to process (0=no)
zOS_SIM equ      0x0f      ; mask for software interrupts (low 3 always==1)

zOS_TOS equ      0x0e      ; STKPTR for full stack (0x0f reserved for ISRs)
zOS_BOS equ      0x0b      ; STKPTR for empty stack (first push is to 0x0c)

;;; bank 0 memory space for managing jobs, 1@0x20, 2@0x30, ... , 5@0x60
zOS_J1M equ      0x20
zOS_J2M equ      0x30
zOS_J3M equ      0x40
zOS_J4M equ      0x50
zOS_J5M equ      0x60

zOS_MEM macro     fsrnum,job,offset
local    fsrn
if (fsrnum & 3)
fsrn set 1
else
fsrn set 0
endif
swapf    job,w      ;inline void zOS_MEM(int8_t* *fsrnum,
addlw    0x10        ; const int8_t* job,
andlw    0x70        ; const
if (offset)
addlw    offset      ; int8_t offset) {
endif
movwf    FSR#v(fsrn)L ; *fsrnum = (((job + 1) & 0x07) << 4) + offset;
clrf     FSR#v(fsrn)H ; } // zOS_MEM()
endm

```

```

;;; macro to wind the circular stack around from the running job# to the new job
;;; (before restoring the new job's STKPTR and copying its return address there)
;;; typically: zOS_ROL BSR_SHAD,JOB_NUM(BSR?),zOS_TMP,FSR0,zOS_STK
;;; note: caller is responsible for making sure the STKPTR/_SHAD bank is active
zOS_ROL macro    old,new,temp,fsrnum,base
    local fsrn,loop1,loop2,done
    if (fsrnum & 3)
fsrn set 1
    else
fsrn set 0
    endif
    movlw    low base        ;inline void zOS_ROL(const int8_t* old,
    movwf    FSR#v(fsrn)L    ;          const int8_t* new,
    movlw    high base       ;          int8_t* temp,
    movwf    FSR#v(fsrn)H    ;          int16_t* *fsrnum,
    movf     new,w           ;          int8_t* base) {
    subwf    old,w           ; //responsibility of caller to banksel STKPTR
    btfsc    STATUS,Z        ; if (*new == *old) // nothing to do
    bra      done            ; return;
    decf     WREG,w          ; w = new - old - 1;
    btfsc    WREG,7          ; // set STKPTR to the current location of the
    addlw    5               ; // stack cell that needs to be rotated into
    movwf    STKPTR          ; // STK_TOP, then record this value in temp for
    lslf     STKPTR,f        ; // comparison to know when to exit the loop
    addwf    STKPTR,w        ; // that copies the entire stack (except 0x0f)
    addlw    2               ; // into 30-byte scratch in the unrolled order
    movwf    STKPTR          ;
    movwf    temp           ; for (STKPTR = *temp = 2+3*((w<0) ? (w+5) : w);

loop1
    movf     TOSL,w          ; STKPTR != *temp + 1;
    movwi    FSR#v(fsrn)++   ; STKPTR = (STKPTR>0) ? (STKPTR-1):zOS_TOS;
    movf     TOSH,w          ;
    movwi    FSR#v(fsrn)++   ; (*fsrnum)++ = (TOSH << 8) | TOSL;
    decf     STKPTR,f        ;
    movlw    zOS_TOS         ;
    btfsc    STKPTR,4        ;
    movwf    STKPTR          ;
    movf     temp,w          ;
    xorwf    STKPTR,w        ;
    btfss    STATUS,Z        ; // now rebuild the unrolled stack
    bra      loop1           ;
    clrf     STKPTR          ; for (STKPTR = 0;

loop2
    moviw    --FSR#v(fsrn)   ; STKPTR <= zOS_TOS;
    movwf    TOSH            ; STKPTR++) {
    moviw    --FSR#v(fsrn)   ; TOSH = *(*fsrnum) >> 8;
    movwf    TOSL            ; TOSL = **--(*fsrnum) & 0x00ff;
    incf     STKPTR,w        ; }
    movwf    STKPTR          ;
    sublw    zOS_TOS         ;
    btfss    WREG,7          ;
    bra      loop2           ; } // zOS_ROL()

done
endm

#ifdef GPASM
zOS_RTL equ    (STATUS_SHAD-FSR1H_SHAD-2)
zOS_RTH equ    (STATUS_SHAD-FSR1H_SHAD-1)
zOS_RTS equ    (STATUS_SHAD-FSR1H_SHAD+2)
#else
zOS_RTL equ    ((STATUS_SHAD-FSR1H_SHAD-2)&0x3f)
zOS_RTH equ    ((STATUS_SHAD-FSR1H_SHAD-1)&0x3f)
zOS_RTS equ    ((STATUS_SHAD-FSR1H_SHAD+2)&0x3f)
#endif

;;; running job#: 1      2      3      4      5
;;; stack pos 15: 3rd(1) 3rd(2) 3rd(3) 3rd(4) 3rd(5)
;;; stack pos 14: 2nd(1) 2nd(2) 2nd(3) 2nd(4) 2nd(5)
;;; stack pos 13: 1st(1) 1st(2) 1st(3) 1st(4) 1st(5)

```

```

;;; stack pos 12: 0th(1) 0th(2) 0th(3) 0th(4) 0th(5)
;;; stack pos 11: 2nd(5) 2nd(1) 2nd(2) 2nd(3) 2nd(4)
;;; stack pos 10: 1st(5) 1st(1) 1st(2) 1st(3) 1st(4)
;;; stack pos 9: 0th(5) 0th(1) 0th(2) 0th(3) 0th(4)
;;; stack pos 8: 2nd(4) 2nd(5) 2nd(1) 2nd(2) 2nd(3)
;;; stack pos 7: 1st(4) 1st(5) 1st(1) 1st(2) 1st(3)
;;; stack pos 6: 0th(4) 0th(5) 0th(1) 0th(2) 0th(3)
;;; stack pos 5: 2nd(3) 2nd(4) 2nd(5) 2nd(1) 2nd(2)
;;; stack pos 4: 1st(3) 1st(4) 1st(5) 1st(1) 1st(2)
;;; stack pos 3: 0th(3) 0th(4) 0th(5) 0th(1) 0th(2)
;;; stack pos 2: 2nd(2) 2nd(3) 2nd(4) 2nd(5) 2nd(1)
;;; stack pos 1: 1st(2) 1st(3) 1st(4) 1st(5) 1st(1)
;;; stack pos 0: 0th(2) 0th(3) 0th(4) 0th(5) 0th(1)

;;; continue with next iteration of HWI-searching loop (mustn't clobber FSR0!)
;;; when searching for the correct hardware interrupt handler, without stack hit
zOS_RET macro
    pagesel zos_nhw
    goto    zos_nhw          ;#define zOS_RET() goto zos_nhw
endm

;;; at the end of any interrupt handler goes back to scheduler without stack hit
zOS_RFI macro
    pagesel zos_noc
    goto    zos_noc          ;inline void zOS_RFI(void) { goto zos_noc; }
endm

zOS_RFS macro    retreg
    pagesel zos_sch          ;inline void zOS_RFS(int8_t* retreg) { //from SWI
    if (retreg-WREG)
        movf    retreg,w      ; w = *retreg; goto zos_sch; //clobbers WREG_SHAD
    endif
    goto    zos_sch          ; } // zOS_RFS()
endm

;;; find something runnable (i.e. PCH != 0, but sleep MSB is OK), at job+/-1
;;; according to incr then branch to unf if job-1 == 0 or job+1 > zOS_NUM,
;;; with fsrnum pointing to job's bank 0 structure and then incremented +/-16
zOS_LIV macro    fsrnum,job,incr,unf
    local fsrn,loop
    if (fsrnum & 3)
fsrn set 1
    else
fsrn set 0
    endif
loop
    if (incr)
        movlw    0x10          ;inline int8_t zOS_LIV(int8_t* *fsrnum,
    else
        movlw    0-0x10        ; uint8_t *job, int8_t incr, void *(*unf)()) {
    endif
    addwf    FSR#v(fsrn)L,f    ; do {
    if (incr)
        incf     job,f          ; *fsrnum += incr ? 0x10 : -0x10; // next struct
        movlw    0xff-zOS_NUM   ; job += incr ? 1 : -1; // next job#
        addwf    job,w          ; if ((job == 0) || (job >= zOS_NUM+1)) { //past
        btfss    WREG,7          ;
    else
        decf     job,f          ; goto unf; // Z was set
        btfsc    STATUS,Z        ; } else if (zOS_PCH[fsrnum]) // found runnable
    endif
    bra      unf              ; return w = zOS_PCH[fsrnum]; // Z was cleared
    moviw    zOS_PCH[FSR#v(fsrn)]
    btfsc    STATUS,Z          ; } while (1); // job is runnable (or unf was 0)
    bra      loop              ; } // zOS_LIV()
endm

#ifdef FSR0
#else

```

```

FSR0     equ     FSR0L
#endif
#ifdef FSR1
#else
FSR1     equ     FSR1L
#endif

;; a job switch is attempted with every incoming interrupt
;; user jobs are responsible for processing their own interrupts
;; with an interrupt handler registered at the time of creation

org      0x0000
pagesel  zos_ini
goto     zos_ini      ;<--zos_ini is run upon reset to bootstrap zOS

org      0x0002
pagesel  zos_swj
goto     zos_swj      ;<--zos_SWI is call to 0x0002, a jump to zos_swj

;; enter handler which will zOS_RFI() to zos_sch if it's the correct one
;; (and we're not still in the bank-0 initialization before interrupts),
;; after clearing the interrupt flag...else zOS_RET() back up to zos_nhw

org      0x0004
;; find first willing handler for an enabled interrupt matching _xIM bit
#ifdef PIE0
zos_PIE  equ     PIE0
#else
zos_PIE  equ     INTCON
#endif
zos_004
movlw    zOS_NUM+1      ;__isr void zos_004(void) {
movwf    zOS_JOB        ; zOS_JOB = zOS_NUM+1; // search from high to low
zos_MEM  FSR0,zOS_JOB,0 ; fsr0 = 0x10 * (1 + zOS_JOB);

zos_nhw
zos_LIV  FSR0,zOS_JOB,0,zos_004
clrwdt   ; do { // until serviceable by running ISR since
banksel  zos_PIE
movlw    zOS_HIM[FSR0] ; int8_t w = 0; // no runnable job schedulable
andwf    zos_PIE,w      ; clrwdt();
btfss    STATUS,Z      ; while (zos_LIV(&fsr0, &zOS_JOB, 0)) {
bra       zos_cmp       ; //match enabled interrupts against HIM fields

#ifdef PIE1
movlw    zOS_HIM[FSR0] ; if ((w = zOS_HIM[fsr0] & zOS_PIE))
banksel  PIE1
andwf    PIE1,w        ; break;
btfss    STATUS,Z      ; if ((w = zOS_HIM[fsr0] & zOS_PIE1))
bra       zos_cmp       ; break;

#endif
#ifdef PIE2
movlw    zOS_HIM[FSR0] ;
andwf    PIE2,w        ;
btfss    STATUS,Z      ; if ((w = zOS_HIM[fsr0] & zOS_PIE2))
bra       zos_cmp       ; break;

#endif
#ifdef PIE3
movlw    zOS_HIM[FSR0] ;
andwf    PIE3,w        ;
btfss    STATUS,Z      ; if ((w = zOS_HIM[fsr0] & zOS_PIE3))
bra       zos_cmp       ; break;

#endif
#ifdef PIE4
movlw    zOS_HIM[FSR0] ;
andwf    PIE4,w        ;
btfss    STATUS,Z      ; if ((w = zOS_HIM[fsr0] & zOS_PIE4))
bra       zos_cmp       ; break;

#endif
#ifdef PIE5
movlw    zOS_HIM[FSR0] ;

```

```

andwf    PIE5,w        ;
btfss    STATUS,Z      ; if ((w = zOS_HIM[fsr0] & zOS_PIE5))
bra       zos_cmp       ; break;

#endif
#ifdef PIE6
movlw    zOS_HIM[FSR0] ;
andwf    PIE6,w        ;
btfss    STATUS,Z      ; if ((w = zOS_HIM[fsr0] & zOS_PIE6))
bra       zos_cmp       ; break;

#endif
#ifdef PIE7
movlw    zOS_HIM[FSR0] ;
andwf    PIE7,w        ;
btfss    STATUS,Z      ; if ((w = zOS_HIM[fsr0] & zOS_PIE7))
bra       zos_cmp       ; break;

#endif
#ifdef PIE8
movlw    zOS_HIM[FSR0] ;
andwf    PIE8,w        ;
btfss    STATUS,Z      ; if ((w = zOS_HIM[fsr0] & zOS_PIE8))
bra       zos_cmp       ; break;

#endif
#ifdef PIE9
movlw    zOS_HIM[FSR0] ;
andwf    PIE9,w        ;
btfss    STATUS,Z      ; if ((w = zOS_HIM[fsr0] & zOS_PIE9))
bra       zos_cmp       ; break; // found a potential handler for any

#endif
bra       zos_nhw      ; } // interrupt flag in this bit position
zos_cmp
clrf     zOS_MSK        ; if (w) {
movlw    zOS_ISH[FSR0] ; zOS_MSK = 0; //indicates HWI (not SWI) type
movwf    PCLATH         ; *(zos_ISR[fsr0]);
movlw    zOS_ISR[FSR0] ; }
movwf    PCL            ; } // if handler refuses, loops to the next job

;; scheduler begins here, called either after HWI/SWI done or zOS_RUN():

zos_sch
banksel  WREG_SHAD
movwf    WREG_SHAD      ;zos_sch: // w sent via zOS_RFS()

zos_noc
banksel  WREG_SHAD
movf     BSR_SHAD,w      ; WREG_SHAD = w;zos_noc://lobber from zOS_RFI()
btfsc    STATUS,Z        ; // stay in _SHAD/STKPTR/TOS bank until retfie
bra       zos_don        ; if ((zos_JOB = BSR_SHAD)!= 0)//2x max or '004
movwf    zOS_JOB        ; for (zos_MSK = 2; zOS_MSK; zOS_MSK--) {
movlw    3                ;
movwf    zOS_MSK        ; //zos_MSK=2 first time through,1 after wrap
bra       zos_lst        ; zOS_MEM(fsr0,zOS_JOB,0);

zos_itr
zos_LIV  FSR0,zOS_JOB,1,zos_wra
clrwdt   ; //zos_LIV leaves PCH in WREG, test runnable?
btfsc    WREG,zOS_WAI    ; while(zos_LIV(fsr0,zOS_JOB,1)&(1<<zOS_WAI))
bra       zos_itr        ; clrwdt();

;; if this point is reached, a runnable job was found with job# zOS_JOB
;; (but we skip a whole bunch of trivial copies if zOS_JOB==BSR_SHAD)
movf     BSR_SHAD,w      ;
xorwf    zOS_JOB,w        ;
btfsc    STATUS,Z        ;
bra       zos_don        ; if (zos_JOB != BSR_SHAD) {

;; copy the interrupted job's (BSR_SHAD) criticals into its bank 0 slot;
;; by pure chance this clobbers the "unused" range 0x72~0x7b on 1st run!
zos_MEM  FSR0,BSR_SHAD,zOS_PCL
movf     TOSL,w          ; fsr0 = 0x10 * (1+BSR_SHAD) + zOS_PCL;
movwi    FSR0++          ; *fsr0++ = TOSL; // return address from IRQ
movf     TOSH,w          ;
movwi    FSR0++          ; *fsr0++ = TOSH;

```

```

movf STATUS_SHAD,w ;
movwi FSR0++ ; *fsr0++ = STATUS_SHAD;
movf WREG_SHAD,w ;
movwi FSR0++ ; *fsr0++ = WREG_SHAD;
movf STKPTR,w ;
movwi FSR0++ ; *fsr0++ = STKPTR; // not BSR_SHAD
movf PCLATH_SHAD,w ;
movwi FSR0++ ; *fsr0++ = PCLATH_SHAD;
movf FSR0L_SHAD,w ;
movwi FSR0++ ; *fsr0++ = FSR0L_SHAD;
movf FSR0H_SHAD,w ;
movwi FSR0++ ; *fsr0++ = FSR0H_SHAD;
movf FSR1L_SHAD,w ;
movwi FSR0++ ; *fsr0++ = FSR1L_SHAD;
movf FSR1H_SHAD,w ;
movwi FSR0++ ; *fsr0++ = FSR1H_SHAD;

;; get stack spun around to where zOS_JOB expects it on return from ISR
zos_rol BSR_SHAD,zOS_JOB,zOS_MSK,FSR1,zOS_STK

;; copy zOS_JOB's criticals out of its bank 0 slot
zos_mem FSR0,zOS_JOB,zOS_SST
moviw FSR0++ ; fsr0 = 0x10 * (1+zOS_JOB) + zOS_SST;
movwf STATUS_SHAD ; STATUS_SHAD = *fsr0++;
moviw FSR0++ ;
movwf WREG_SHAD ; WREG_SHAD = *fsr0++;
movf zOS_JOB,w ; //point to correct 80-byte local SRAM page
movwf BSR_SHAD ; BSR_SHAD = zOS_JOB; // not STKPTR
movwf ++FSR0 ; //^^ notice BSR = zOS_JOB upon retfie! ^^
movwf PCLATH_SHAD ; PCLATH_SHAD = **++fsr0;
moviw ++FSR0 ;
movwf FSR0L_SHAD ; FSR0L_SHAD = **++fsr0;
moviw ++FSR0 ;
movwf FSR0H_SHAD ; FSR0H_SHAD = **++fsr0;
moviw ++FSR0 ;
movwf FSR1L_SHAD ; FSR1L_SHAD = **++fsr0;
moviw ++FSR0 ;
movwf FSR1H_SHAD ; FSR1H_SHAD = **++fsr0;

;; set new job stack pointer, last step before completing context switch
moviw zOS_RTS[FSR0] ;
movwf STKPTR ; STKPTR = zOS_SSP[FSR0-11];
moviw zOS_RTL[FSR0] ; TOSL = zOS_PCL[FSR0-11];
movwf TOSL ; TOSH = zOS_PCH[FSR0-11];
moviw zOS_RTH[FSR0] ; return (void)__isr;
movwf TOSH ; }

zos_don retfie ; //if this point is reached, search wrapped:
zos_wra clrf zOS_JOB ; fsr0 = 0x10 * (1 + (zOS_JOB = 0));
zos_1st zOS_MEM FSR0,zOS_JOB,0 ; } // wrap around only once, else wait for IRQ
decfsz zOS_MSK,f ; } while (1); // (since no job is schedulable)
bra zos_itr ; } // zOS_004()
bra zos_004 ;int8_t zos_swj(int8_t w){ // call vector at 002

;; software interrupt processing reached by jumping to 0x0002 with W set
;; which then calls to zos_swj, or by jumping to zos_skp after already
;; processing a previous interrupt (since there is only 1 level of SHAD)
;; to skip the copy into the shadow registers
zos_skp movwf zOS_MSK ;
bra zos_sk2 ;

zos_swj ; save the shadow registers (for the ones that have them) to use retfie
bcf INTCON,GIE ; INTCON &= ~(1<<GIE); // interrupt would be bad
movwf zOS_MSK ; zOS_MSK = WREG; // the software interrupt type
movf STATUS,w ;

movwf zOS_JOB ; // only convenient temporary global for STATUS
movf BSR,w ;
banksel BSR_SHAD ; // BSR = the job# that made the interrupt call
movwf BSR_SHAD ; BSR_SHAD = BSR;
movf zOS_JOB,w ;
movwf STATUS_SHAD ; STATUS_SHAD = zos_job = STATUS;
movf PCLATH,w ;
movwf PCLATH_SHAD ; PCLATH_SHAD = PCLATH;
movf FSR0L,w ;
movwf FSR0L_SHAD ; FSR0L_SHAD = FSR0L;
movf FSR0H,w ;
movwf FSR0H_SHAD ; FSR0H_SHAD = FSR0H;
movf FSR1L,w ;
movwf FSR1L_SHAD ; FSR1L_SHAD = FSR1L;
movf FSR1H,w ;
movwf FSR1H_SHAD ; FSR1H_SHAD = FSR1H;

zos_sk2 ; see if the interrupt type is a system one (<8)
pagesel zos_swh
movlw zOS_SI7|zos_SI6|zos_SI5|zos_SI4|zos_SI3
andwf zOS_MSK,w ; if (0 == /* call-type number: */ WREG_SHAD &
btfss STATUS,Z ; (zos_SI7|zos_SI6|zos_SI5|zos_SI4|zos_SI3)) {
goto zos_swh ; // handle a system zOS_SWI call:

;; zOS_NEW requires us to search for a BSR value first among empty slots
movf BSR_SHAD,w ;
movwf BSR ; // BSR unchanged from what it had been at call
movf zOS_MSK,f ;
btfss STATUS,Z ; if (zos_MSK == zOS_NEW /*==0*/) {
bra zos_swp ; zos_cre:

zos_cre clrf zOS_JOB ; zos_job = 0;
zos_mem FSR1,zOS_JOB,0

zos_emp movlw 0x10 ; for (fsr1 = 0x10*(1+zos_job);
addwf FSR1L,f ;
incf zOS_JOB,f ; zos_job++ <= zOS_NUM;
movlw 0xff-zOS_NUM ;
addwf zOS_JOB,w ;
btfsc STATUS,Z ; fsr1 += 0x10) {
bra zos_err ; if (zos_PCH[FSR1] == 0)
moviw zOS_PCH[FSR1] ; break;
btfss STATUS,Z ; }
bra zos_emp ; if (zos_job <= zOS_NUM) {

zos_dup movf FSR0L,w ; // save handle now so we can re-use fsr0
movwi zOS_HDL[FSR1] ; // (no harm if we don't validate it as PCH)
movf FSR0H,w ; zOS_HDL[fsr1] = fsr0 & 0x00ff;
movwi zOS_HDH[FSR1] ; zOS_HDH[fsr1] = fsr0 >> 8;
movf BSR,f ; if (bsr == 0)
btfsc STATUS,Z ; goto zos_swk; // job#0 (launcher) has perm
bra zos_swk ; fsr0 = 0x10 * (1+bsr); // struct for caller
zos_mem FSR0,BSR,0
moviw zOS_HDH[FSR0] ; if (zos_HDH[fsr0] & (1<<zOS_PRB))
btfsc WREG,zOS_PRB ; goto zos_swk; // job has privileged perms
bra zos_swk ; }

zos_err clrf zOS_JOB ; zos_job = 0;
zos_rfs zOS_JOB ; zOS_RFS(zOS_JOB); // perms error or no empty

;; see if we're not running inside a job context (1 <= job# <= zOS_NUM)
;; in which case need to grab the targeted job from AR0 (if not zOS_NEW)
;; or find a targetable slot (if zOS_NEW)
;; unprivileged jobs can only do most things to themselves

zos_swp movf BSR,w ; } else {
movwf zOS_JOB ; zos_job = bsr;
btfsc STATUS,Z ; if (bsr != 0) {
bra zos_elv ; fsr1 = 0x10 * (1+bsr); // struct for job

```

```

        zOS_MEM FSR1,BSR,0
        movlw   zOS_HDH[FSR1] ; if (zOS_HDH[fsr1] & (1<<zOS_PRB) == 0)
        btfs    WREG,zOS_PRB ; goto zos_swk; // disallowed job in zOS_AR0
        bra     zos_swk ; }

        ;; desired job# (instead of this one) into BSR from AR0 (if not zOS_NEW)
zos_elv
        movf    zOS_AR0,w ; // access granted, bring the patient to me
        movwf   BSR ; bsr = zOS_AR0;
        zOS_MEM FSR1,BSR,0

zos_swk
        movf    zOS_MSK,w ; }
        brw     ; switch (zOS_MSK) { // guaranteed < 8
        bra     zos_sw0 ;
        bra     zos_sw1 ;
        bra     zos_sw2 ;
        bra     zos_sw3 ;
        bra     zos_sw4 ;
        bra     zos_sw5 ;
        bra     zos_sw6 ;
        bra     zos_sw7 ; case zOS_NEW:

zos_sw0
        movf    zOS_AR0,w ;
        movwi   zOS_ISR[FSR1] ; zOS_ISR[fsr1] = zOS_AR0;
        movf    zOS_AR1,w ;
        movwi   zOS_ISH[FSR1] ; zOS_ISH[fsr1] = zOS_AR1;
        movf    zOS_AR2,w ;
        movwi   zOS_HIM[FSR1] ; zOS_HIM[fsr1] = zOS_AR2;
        movf    zOS_AR3,w ;
        movwi   zOS_SIM[FSR1] ; zOS_SIM[fsr1] = zOS_AR3;
        bra     zos_sw3 ; goto zos_sw3;

zos_sw1
        moviw   zOS_PCH[FSR1] ; case zOS_SLP:
        iorlw   0x80 ; zOS_PCH[fsr1] |= 0x80;
        movwi   zOS_PCH[FSR1] ; zOS_RFS(zOS_JOB);
        zOS_RFS zOS_JOB

zos_sw2
        clrw    ; case zOS_END: zOS_PCH[fsr1] = 0;
        movwi   zOS_PCH[FSR1] ; zOS_RFS(zOS_JOB); // killing is so quick
        zOS_RFS zOS_JOB

zos_sw3
        moviw   zOS_HDL[FSR1] ; case zOS_RST: zos_sw3:
        movwi   zOS_PCL[FSR1] ; // retain HDL MSB (which indicate privilege)
        moviw   zOS_HDH[FSR1] ; zOS_PCL[fsr1] = zOS_HDL[fsr1];
        andlw   0x7f ; // clear PC MSB (which indicates sleepiness)
        movwi   zOS_PCH[FSR1] ; zOS_PCH[fsr1] = zOS_HDH[fsr1] & 0x7f;
        movlw   zOS_BOS ; zOS_SSP[fsr1] = zOS_BOS;
        movwi   zOS_SSP[FSR1] ;

        lslf    zOS_JOB,w ;
        iorlw   0x70 ;
        movwf   FSR1L ; fsr1 = 0x70 | (zOS_JOB << 1);
        clrw    ; 0[fsr1] = 1[fsr1] = 0; // mailbox guar'ed 0
        movwi   0[FSR1] ; case zOS_YLD:
        movwi   1[FSR1] ; zOS_RFS(zOS_JOB);

zos_sw4

#ifdef zOS_MIN
zos_sw5
zos_sw6
zos_sw7
        zOS_RFS zOS_JOB
#else
        zOS_RFS zOS_JOB

zos_sw5
        ;; copy job BSR's 0x20-0x6f into every non-running bank first
        clrf    FSR1L ; case zOS_FRK:
        clrf    FSR1H ; fsr1 = 1 << 7;
        clrf    zOS_JOB ; for (zos_job = 1;

```

```

zos_cpl
        movlw   0x80 ; zos_job++ <= zOS_NUM; fsr1 += 0x80) {
        andwf   FSR1L,f ; fsr1 &= 0xff80;
        addwf   FSR1L,f ;
        clrw    ;
        addwfc   FSR1H,f ; fsr1 += 0x80;
        incf    zOS_JOB,f ;
        movlw   0xff-zOS_NUM ;
        addwf   zOS_JOB,w ;
        btfs    STATUS,Z ;
        bra     zos_cpd ;

        zOS_MEM FSR0,BSR,0
        moviw   zOS_PCH[FSR0] ; fsr0 = 0x10 * (1+BSR);
        btfs    STATUS,Z ; if (zOS_PCH[fsr0] == 0)
        bra     zos_cpl ; continue; // can't touch a running job

        lsrf    BSR,w ;
        movwf   FSR0H ;
        clrf    FSR0L ;
        rrf     FSR0L,f ;
        movlw   0x6f ;
        iorwf   FSR0L,f ; fsr0 = (BSR << 7) | 0x6f;
        iorwf   FSR1L,f ; for (fsr1 |= 0x6f; fsr1 & 0x7f >= 0x20;

zos_cp2
        moviw   FSR0-- ;
        movwi   FSR1-- ; *fsr1-- = *fsr0--
        movlw   0x60 ;
        andwf   FSR0L,w ;
        btfs    STATUS,Z ;
        bra     zos_cp2 ;
        bra     zos_cpl ; }

zos_cpd
        ;; now copy job BSR's bank0 struct to the zOS_AR registers and zOS_NEW()
        ;;;FIXME: should copy the rest of state, i.e. memory variables to be a true fork
        ;;;FIXME: disallow fork if any HWI is defined for the process (assume conflicts)
        movf    BSR,w ;
        movwf   zOS_JOB ; zOS_JOB = BSR;
        zOS_MEM FSR1,zOS_JOB,0
        moviw   zOS_PCH[FSR1] ;
        btfs    STATUS,Z ;
        bra     zos_sw4 ; if (zOS_PCH[fsr1]) {
        moviw   zOS_HDL[FSR1] ;
        movwf   FSR0L ;
        moviw   zOS_HDH[FSR1] ;
        movwf   FSR0H ; fsr0 = (zOS_HDH[fsr1]<<8) | zOS_HDL[fsr1];
        moviw   zOS_ISR[FSR1] ;
        movwf   zOS_AR0 ; zOS_AR0 = zOS_ISR[fsr1];
        moviw   zOS_ISH[FSR1] ;
        movwf   zOS_AR1 ; zOS_AR1 = zOS_ISH[fsr1];
        moviw   zOS_HIM[FSR1] ;
        movwf   zOS_AR2 ; zOS_AR2 = zOS_HIM[fsr1];
        moviw   zOS_SIM[FSR1] ;
        movwf   zOS_AR3 ; zOS_AR3 = zOS_SIM[fsr1];
        banksel WREG_SHAD ;
        clrf    WREG_SHAD ; WREG_SHAD = zOS_NEW;
        movlb   0 ; goto zos_cre;//spooof privilege to fork self
        bra     zos_cre ; }

zos_sw6
        movf    BSR,w ; case zOS_EXE:
        movwf   zOS_JOB ; zOS_JOB = BSR;
        zOS_MEM FSR1,zOS_JOB,0
        banksel WREG_SHAD ; fsr1 = 0x10 * (1+zOS_JOB);
        clrf    WREG_SHAD ; WREG_SHAD = zOS_NEW;
        movlb   0 ; //spooof privilege to overwrite
        bra     zos_dup ; goto zos_dup;

zos_sw7
        movf    zOS_AR2,w ; case zOS_FND:

```

```

        btfss STATUS,Z      ;
        movlw zOS_NUM       ;
        addlw 1             ;
        movwf zOS_JOB       ;
        addlw 0xfe-zOS_NUM   ; if (zOS_AR2 && ((uint8_t)zOS_AR2<=zOS_NUM))
        btfss WREG,7        ; zOS_JOB = zOS_AR2 + 1;
        movlw 1+zOS_NUM     ; else
        movwf zOS_JOB       ; zOS_JOB = zOS_NUM + 1;
        zOS_MEM FSR1,zOS_JOB,0 ; fsr1 = 0x10 * (1 + zOS_JOB);

zos_nxt
        zOS_LIV FSR1,zOS_JOB,0,zos_bad
        moviw zOS_HDL[FSR1] ; while (zOS_LIV(&fsr1, &zOS_JOB, 0)) {
        xorwf zOS_AR0,w     ;
        btfss STATUS,Z     ;
        bra zos_nxt        ;
        moviw zOS_HDH[FSR1] ; void (*a)() = (zOS_AR1<<8)|zOS_AR0;
        xorwf zOS_AR1,w     ; void (*b)() = (zOS_HDH[fsr1]<<8)|zOS_HDL[fsr1]
;
        andlw 0x7f         ;
        btfss STATUS,Z     ; if (a & 0x7f == b & 0x7f)
        bra zos_nxt        ; zOS_RFS(zOS_JOB);
        zOS_RFS zOS_JOB    ; }

zos_bad
        clrw               ;
        zOS_RFS WREG        ; zOS_RFS(w = 0);

#endif

; ; else handle the software interrupt with the first registered handler

zos_swh
        banksel BSR_SHAD
        incf BSR_SHAD,w     ; // a swi number of 0xff is special now, will
        incfsz zOS_MSK,f    ; // cause the calling job to invoke its own
        movlw 1+zOS_NUM     ; // handler without knowledge of its SWI code!
        decf zOS_MSK,f      ; // (at the cost of 4 extra instruction cycles)
        movwf zOS_JOB       ; zos_job =1+((zos_msk==0xff)?BSR_SHAD:zOS_NUM);

        zOS_MEM FSR0,zOS_JOB,0 ; while (zOS_LIV(&fsr0, &zOS_JOB, 0)) { //search

zos_sw1
        zOS_LIV FSR0,zOS_JOB,0,zos_swm
        moviw zOS_SIM[FSR0] ;
        andwf zOS_MSK,w     ;
        btfsc STATUS,Z     ;
        bra zos_sw1        ; if ((zos_msk & zOS_SIM[fsr0]) != 0) { //found
        movwf zOS_MSK       ; zos_msk &= zOS_SIM[fsr0];
        moviw zOS_ISH[FSR0] ; goto (void*)(zOS_ISR[fsr0]); // will zOS_RFS
        movwf PCLATH        ; }
        moviw zOS_ISR[FSR0] ; }
        movwf PCL           ; zOS_RFS(WREG = 0);

; ; no registered SWI handler: jump into the hardware interrupt scheduler

zos_swm
        zOS_RFS WREG

zos_ini
; ; clear out page 0 to reflect no running tasks, set global data to 0's
        movlb 0             ; "invalid" job# used to get perms for zOS_NEW
        movlw 0x7f          ; bsr = 0;
        movwf FSR0L         ;
        clrf FSR0H          ; for (fsr0 = 0x007f; fsr >= 0x0020; fsr--)

zos_zer
        clrw               ;
        movwi FSR0--        ; *fsr = 0; // only zOS_PCH is critical
        movlw 0x60          ;
        andwf FSR0L,w       ;
        btfss STATUS,Z     ;
        bra zos_zer        ;

; ; your program starts here, with a series of launcher instructions for
; ; 1) setting up oscillators, timers, other peripherals, etc.

```

```

; ; (with the appropriate and ineviatable bank switching)
; ; 2) starting jobs with calls to zOS_NEW or its zOS_LAU wrapper
; ; (being sure to stay in bank 0 or using job macros zOS_CON/zos_MON)
; ; 3) calling zOS_RUN (which will enable interrupts) to start job 1

```



```

;;; zosmacro.inc
;;; potentially useful (but not mandatory) macros for zOS
;;;
;;; total memory footprint (for a PIC16F1847, including the zOS base):
;;; no memory words used upon inclusion (before expansion of a macro)
;;; ~256 14-bit words if only zOS_CON() job is started to buffer console output
;;; _?_ 14-bit words for full-featured monitor zOS_MON()
;;; _?_ 14-bit words for job manager shell zOS_MAN()

#ifdef UCFG
#define zOS_ME BSR,w : xorlw 0x8; // advance zOS use past DPSRAM; FIXME:untested
#else
#define zOS_ME BSR,w ; // "movf/andwf/xorwf zOS_ME" can't clobber BSR
#endif

zOS_GLO macro fsrnum,job
    local fsrn
    if (fsrnum & 3)
fsrn set 1
    else
fsrn set 0
    endif
    if (job)
        lslf job,w ;inline void zOS_GLO(int8_t**fsrnum,int8_t*job){
    else
        lslf zOS_ME ;
    endif
    andlw 0x0e ; int8_t w = 0x70 | ((job ? *job : bsr) << 1);
    iorlw 0x70 ;
    movwf FSR#v(fsrn)L ;// documentation suggests 5 but BSR now 6 bits!
    movlw 0x1f ; *fsrnum = (*fsrnum & 0x1f00) | w;
    andwf FSR#v(fsrn)H,f ;} // zOS_GLO()
endm

zOS_MY2 macro fsrnum ;inline int8_t zOS_MY2(int8_t**fsrnum){
    zOS_GLO fsrnum,0 ; return zOS_GLO(fsrnum, 0);
endm ;} // zOS_MY2()

zOS_LOC macro fsrnum,job,offset
    local fsrn
    if (fsrnum & 3)
fsrn set 1
    else
fsrn set 0
    endif
    if (offset)
        movlw offset<<1 ;inline int8_t zOS_LOC(int8_t* *fsrnum,
        movwf FSR#v(fsrn)L ; int8_t* job, uint8_t offset) {
    else
        clrf FSR#v(fsrn)L ;
    endif
    if (job - FSR#v(fsrn)H)
        lsrif job,w ;
        movwf FSR#v(fsrn)H ; return (*fsrnum = (job<<7) | offset) >> 8;
    else
        lsrif job,f
    endif
    rrf FSR#v(fsrn)L,f ;} // zOS_LOC()
endm

zOS_ADR macro adr,msb
    movlw low adr ;inline void zOS_ADR(void* a) {
    movwf FSR0L ; if (msb) fsr0 = 0x8000 | a;
    movlw high adr ; else fsr0 = 0x7fff & a;
    movwf FSR0H ;} // zOS_ADR()
    if (msb)
        bsf FSR0H,7
    else
        bcf FSR0H,7

```

```

endif
endm

zOS_INT macro lhw,lsw
    if (lhw|lsw)
        movf FSR0L,w ;inline void zOS_INT(const lhw, const lsw) {
        zOS_ARG 0
        movf FSR0H,w ; if (lhw == 0 && lsw == 0) fsr0 = 0;
        zOS_ARG 1
        movlw lhw ; zOS_ARG(0, fsr0 & 0x00ff);
        zOS_ARG 2
        movlw lsw ; zOS_ARG(1, fsr0 >> 8);
        zOS_ARG 3
        else
            clrw ; zOS_ARG(2, lhw);
            movwf FSR0L ; zOS_ARG(3, lsw);
            movwf FSR0H ;} // zOS_INT()
        zOS_ARG 0
        zOS_ARG 1
        zOS_ARG 2
        zOS_ARG 3
    endif
endm

zOS_SWI macro type ;inline void zOS_SWI(const int8_t type) {
    movlw type ;
    movlp 0x00 ; zos_swj(type);
    call 0x02 ;} // zOS_SWI()
endm

zOS_TAI macro type ;inline void zOS_TAI(const int8_t type) {
    movlw type ; w = type; goto zos_skp;
    pagesel zos_skp
    goto zos_skp ;} // zOS_TAI()
endm

zOS_LAU macro stash ;inline void zOS_LAU(int8_t* stash) {
    local retry

retry
    zOS_SWI zOS_NEW
    movf WREG,w ; do {
    btfsc STATUS,Z ; w = zOS_SWI(zOS_NEW);
    bra retry ; } while (w == 0);
    if (stash - WREG)
        movwf stash ; *stash = w;
    endif
endm ;} // zOS_LAU()

zOS_INI macro fsrnum,val0,vall
    if (fsrnum & 3)
        set 1
    else
        set 0
    endif
;after: zOS_LAU FSR#v(fsrn)L
    lslf FSR#v(fsrn)L,f ;inline void zOS_INI(uint8_t* fsrnum, uint8_t
    movlw 0x70 ; val0, uint8_t vall) {
    iorwf FSR#v(fsrn)L,f ; //fsrnum starts and ends as a launched job#
    clrf FSR#v(fsrn)H ; fsrnum = 0x70 | (fsrnum << 1);
    movlw val0 ; // change global mailbox to non-0 if desired
    movwi FSR#v(fsrn)++ ; fsrnum[0] = val0;
    movlw vall ;
    movwi FSR#v(fsrn)-- ; fsrnum[1] = vall;
    lsrif FSR#v(fsrn),w ; fsrnum = (fsrnum >> 1) & 0x07; // unchanged
    andlw 0x07 ;}

zOS_DIS macro fsrnum,job ;inline void zOS_DIS(int8_t* *fsr, int8_t job) {

```

```

    if (fsrnum & 3)
fsrn    set 1
    else
fsrn    set 0
    endif
    if (job)
        zOS_MEM FSR#v(fsrn),job,zOS_HDH ; *fsr = 0x10 * (1+job) + zOS_HDH;//priv
        btfsc INDF#v(fsrn),zOS_PRB ; if (**fsr & (1<<zOS_PRB))
    endif
    bcf INTCON,GIE ; INTCON &= ~(1<<GIE);
    endm ;} // zOS_DIS()

zOS_ENA macro ;inline void zOS_ENA(void) {
    bsf INTCON,GIE ; INTCON |= 1<<GIE;
    endm ;} // zOS_ENA()

zOS_ARG macro arg
    local num
num set (arg & 0x03)
    if (num == 0)
        bcf INTCON,GIE ;inline void zOS_ARG(const int8_t arg, int8_t w)
    endif
    movwf zOS_AR#v(num) ;{if (!arg) INTCON &= ~(1<<GIE); zOS_AR0[arg]=w;}
    endm

zOS_RUN macro t0enable,t0flags
    ; start a TMR0 interrupt since none found (most in INTCON, others PIE0)
    local boot
zOS_T0E equ t0enable
zOS_T0F equ t0flags
    if (zOS_T0E)
        banksel zOS_T0E
        bsf zOS_T0E,T0IE ;inline void zOS_RUN(uint8_t* t0enable) {
            if (zOS_T0E - INTCON)
                bsf INTCON,PEIE ; if (t0enable) { *t0enable |= 1<<T0IE;
            endif
        }
        ; advance the stack pointer to allow 5 stacks of 3 each (+1 if running)
        banksel STKPTR ; if (t0enable != INTCON) INTCON |= 1<<PEIE;
        movlw zOS_BOS ; }
        movwf STKPTR ; STKPTR = zOS_BOS; // every job bottom of stack

        ; set the active job to the first (and potentially only), interrupts ON
        movlw 1+zOS_NUM ; bsr_shad = w = 1+zOS_NUM; // will wrap around
        movwf BSR_SHAD ; boot(); // run the scheduler to grab its PC
        pagesel boot ;} // zOS_RUN()
        call boot ;

    boot
        bsf INTCON,GIE ;void boot(void) { INTCON |= 1<<GIE; zOS_RFI();}
        zOS_RFI
    endm

zOS_DBG macro
    local loop
    banksel STKPTR
    clrf STKPTR ;inline void zOS_DBG(void) {
    clrw ; for (int8_t w = STKPTR = 0;

loop
    clrf TOSH ; w < 16; w++){
    movwf TOSL ; TOSH = 0;
    incf STKPTR,w ; TOSH = w;
    andlw 0x0f ;
    movwf STKPTR ; STKPTR = (STKPTR + 1) % 16;
    btfss STATUS,Z ; }
    bra loop ; STKPTR = -1;
    decf STKPTR,f ; // still in job "0"
    movlb 0 ;} // zOS_DBG()
    endm

```

```

#ifdef PID1CON
    ; 16x16bit signed multiply zOS_AR1:0 * zOS_AR3:2, core yielded during 7ms math
    zOS_MUL macro fsrnum
        local fn,inout,fac0L,fac0H,fac1L,fac1H,zeroH,start,con,setup,enb,bsy
        if (fsrnum & 3)
            set 1
        else
            set 0
        endif
        inout set 0x1f80 & PID1SETL
        fac0L set 0x1f & PID1K1L
        fac0H set 0x1f & PID1K1H
        fac1L set 0x1f & PID1SETL
        fac1H set 0x1f & PID1SETH
        zeroH set 0x1f & PID1INH
        start set 0x1f & PID1INL
        con set 0x1f & PID1CON
        out0 set 0x1f & PID1OUTLL
        out1 set 0x1f & PID1OUTLH
        out2 set 0x1f & PID1OUTHL
        out3 set 0x1f & PID1OUTHH
        setup set (1<<PID1MODEL)
        enb set PID1EN
        bsy set PID1BUSY

        movlw low PID1CON ;void zOS_MUL(int16_t** fsr) {
        movwf FSR#v(fn)L ; *fsr = &PID1CON;
        movlw high PID1CON ;
        movwf FSR#v(fn)H ; do {

    spinget
        btfss INDF#v(fn),enb ; while ((**fsr&(1<<enb))&& // MATHACC for sure
        bra notbusy ; (**fsr&(1<<bsy))) // ours if not busy
        btfss INDF#v(fn),bsy ; // or never enabled
        bra notbusy ;
        zOS_SWI zOS_YLD ; zOS_SWI(zOS_YLD);
        bra spinget ; // interrupts now enabled if zOS_SWI called

    notbusy
        bcf INTCON,GIE ; INTCON &= ~(1<<GIE);
        btfsc INDF#v(fn),enb ; // begin critical section (seizing MATHACC)
        bra spinget ;
        bsf INDF#v(fn),bsy ;
        bra spinget ; } while ((**fsr&(1<<enb))||(**fsr&(1<<bsy)));
        movlw setup ;
        movwf indf#v(fn) ; **fsr = 1<<PID1MODEL; // unsigned mult no accum
        bsf indf#v(fn),enb ; **fsr |= 1<<PID1EN; // selected, then enabled
        movlw low inout ;
        movwf FSR#v(fn)L ;
        movlw high inout ;
        movwf FSR#v(fn)H ; *fsr = &PID1SETL & 0x1f80; // just bank bits
        movf zOS_AR3,w ;
        movwi fac0H[FSR#v(fn)]; (0x1f & PID1K1H)[*fsr] = zOS_AR3;
        movf zOS_AR2,w ;
        movwi fac0L[FSR#v(fn)]; (0x1f & PID1K1L)[*fsr] = zOS_AR2;
        movf zOS_AR1,w ;
        movwi fac1H[FSR#v(fn)]; (0x1f & PID1SETH)[*fsr] = zOS_AR1;
        movf zOS_AR0,w ;
        movwi fac1L[FSR#v(fn)]; (0x1f & PID1SETL)[*fsr] = zOS_AR0;
        clrw ; (0x1f & PID1INH)[*fsr] = 0;
        movwi zeroH[FSR#v(fn)]; (0x1f & PID1INL)[*fsr] = 0; // start multiply
        movwi start[FSR#v(fn)]; // end critical section (seizing MATHACC)
        bsf INTCON,GIE ; INTCON |= 1<<GIE;
        movlw low PID1CON ;
        movwf FSR#v(fn)L ;
        movlw high PID1CON ; *fsr = &PID1CON;
        movwf FSR#v(fn)H ; do {

    spinmul
    #if 0
        clrwldt ; clrwldt();
    #endif

```

```

zos_SWI zos_YLD
btfss   INDF#v(fn),bsy ; zOS_YLD();
bra     spinmul        ; } while (**fsr & 1<<PID1BUSY);
bcf     INTCN,GIE      ; INTCN &= ~(1<<GIE);
bcf     INDF#v(fn),enb ; // begin critical section (copying result)
movlw   low inout      ; **fsr &= ~(1<<enb); // disable MathACC to free
movwf   FSR#v(fn)L     ;
movlw   high inout     ;
movwf   FSR#v(fn)H     ; *fsr = &PID1SETL & 0x1f80; // just bank bits
moviw   out3[FSR#v(fn)] ; zOS_AR3 = (0x1f & PID1OUTTH)[*fsr];
movwf   zOS_AR3        ;
moviw   out2[FSR#v(fn)] ; zOS_AR2 = (0x1f & PID1OUTHL)[*fsr];
movwf   zOS_AR2        ;
moviw   out1[FSR#v(fn)] ; zOS_AR1 = (0x1f & PID1OUTLH)[*fsr];
movwf   zOS_AR1        ;
moviw   out0[FSR#v(fn)] ; zOS_AR0 = (0x1f & PID1OUTLL)[*fsr];
movwf   zOS_AR0        ; // end critical section (when ARx copy's done)
;; bsf   INTCN,GIE      ;} // zOS_MUL()
endm

```

#endif

```

zos_PAG macro   fsrnum
local   fsrn
if (fsrnum & 3)
fsrn set 1
else
fsrn set 0
endif

```

```

swapf   FSR#v(fsrn)L,w ;uint8_t zOS_PAG(void* fsrnum) {
andlw   0x0f           ;
bcf     FSR#v(fsrn)H,5 ;
swapf   FSR#v(fsrn)H,f ;
iorwf   FSR#v(fsrn)H,w ;
swapf   FSR#v(fsrn)H,f ; return w = (fsrnum >> 4);
bsf     FSR#v(fsrn)H,5 ;} // zOS_PAG()
endm

```

```

zos_PTR macro   fsrnum
local   fsrn
if (fsrnum & 3)
fsrn set 1
else
fsrn set 0
endif

```

```

swapf   WREG,w         ;void zOS_PTR(void** fsrnum, uint8_t w) {
movwf   FSR#v(fsrn)H   ;
movwf   FSR#v(fsrn)L   ;
movlw   0x0f           ;
andwf   FSR#v(fsrn)H,f ;
bsf     FSR#v(fsrn)H,4 ;
movlw   0xf0           ; *fsrnum = 0x2000 | w<<4;
andwf   FSR#v(fsrn)L,f ;} // zOS_PTR()
endm

```

```

;;; must be defined with 2 SWI flags: one for malloc(), a different for free()
;;; (typically instantiated with base=0x2210, size = memory size - base)
;;; SWI behavior for malloc(w) is to return pointer in w of 2 middle nybbles
;;; in linear address space, e.g. 0x21 for first cell on a 5-job system, or 0
;;; in w if no free memory of size zOS_AR0*16 bytes was available
;;; SWI behavior for free(w) is to return in w the number of bytes now free/16
;;; intersecting with the address whose middle nybble is zOS_AR0, or 0 in w if
;;; zOS_AR0 didn't point to a valid (i.e. previously allocated) block of bytes
;;;
;;; FIXME: demo idea would be two heap allocators running for two differently
;;; targeted (quantum) allocation heaps, leaving final SWI remaining for zOS_CON
zos_HEA macro   base,size,m,fi ;void zOS_HEA(void* base, void* size, uint8_t
local   isr,decl,task ; mi/*malloc*/,uint8_t fi/*free*/) {

```

```

bra     decl            ; goto decl;

```

```

local   maxnon0,allocated,always0,temp,adrrary,tblsize
local   tblrows,sizarry,memroun,mem3nyb,membase,memsize
maxnon0 set 0x6c
allocated set 0x6d
always0 set 0x6e
temp set 0x6f
adrrary set 0x20
tblsize set 0x50
tblrows set tblsize/2
sizarry set adrrary+tblrows
memroun set base+0xf
mem3nyb set memroun&0xffff
membase set mem3nyb>>4
memsize set size>>4

```

```

isr
local   mloop,mcandid,mexact,mnotall,groloop
local   free,floop,ffound,invalid,done

```

```

movf    zOS_JOB,w      ; isr:
movwf   BSR            ; bsr = zOS_JOB;

```

```

zos_MY2 FSR1           ; fsrl = 0x70|(bsr<<1);
moviw   FSR1++         ;
iorwf   INDF1,w        ;
btfsc   STATUS,Z       ; if (0[fsrl] | 1[fsrl])
bra     invalid        ; goto invalid; // not init'ed according to mbox

```

```

#if (mi - fi)
movf    zOS_MSK,w      ;
andlw   mi             ; ////////////////////////////////////////////////////
btfsc   STATUS,Z       ; ////////////////////////////////////////////////////
bra     free           ; if ((mi != fi) && (zOS_MSK & mi)) ||

```

```

#else
movf    zOS_AR1,w      ; ((mi == fi) && (zOS_AR0!=*sic*/zOS_AR1)) {
movf    zOS_AR0,f      ; // can either assign separate SWIs for malloc
movwf   zOS_AR0        ; // and free or if nearing the SWI limit of 5,
btfsc   STATUS,Z       ; // put the parameter in ARG1 instead of ARG0
bra     free           ; // and ARG0!=0 for malloc() or ==0 for free()

```

```

#endif
zos_LOC FSR0,BSR,adrrary; for (fsr0 = (bsr<<7)+adrrary,
zos_LOC FSR1,BSR,sizarry; fsrl = (bsr<<7)+sizarry;

```

```

mloop
moviw   FSR0++         ; (allocated = temp = *fsr0++); // next poss.
btfsc   STATUS,Z       ; fsrl++) {
bra     invalid        ;
movwf   temp           ;
movwf   allocated      ;
moviw   FSR1++         ; w = *fsrl++; // number of bytes used,0=freed
btfsc   STATUS,Z       ;
bra     mcandid        ; if (w == 0) { // allocatable
bra     mloop          ;

```

```

mcandid
moviw   0[FSR0]        ; w = *fsr0; // upper limit to allocating here
btfsc   STATUS,Z       ; if (w == 0)
bra     invalid        ; goto invalid; // past the highest address

bsf     STATUS,C       ; // temp is now the address of this candidate
comf    temp,f         ; // w is now the next address past candidate
addwfc  temp,w         ;
movwf   temp           ;
subwf   zOS_AR0,w      ; else if ((w = zOS_AR0 - (temp = w-temp))>0)
btfsc   STATUS,Z       ;
bra     mexact         ; // -w now holds extra space beyond requested
btfss   WREG,7         ; // temp now holds total available at allocated
bra     mloop          ;

```

```

bra      mnotall      ;      continue; // not enough allocatable here

mexact
movf     zOS_AR0,w     ;      if (w == 0) { // exactly enough!
movwi    -1[FSR1]     ;      w = -1[fsr1] = zOS_AR0;
bra      done          ;      goto done;

mnotall
movf     maxnon0,f     ;      } else if (adrrary[tblrows-2] != 0) // full
btfss    STATUS,Z     ;      goto invalid;
bra      invalid      ;

movf     zOS_AR0,w     ;      // w == addr to insert, temp == size to insert
movwi    -1[FSR1]     ;      -1[fsr1] = zOS_AR0; // record it as granted
clr      temp         ;      temp = 0;
addwf    allocated,w   ;      for (w = -1[fsr0] + temp; *fsr0; fsr0++,fsr1++
) {
groloop
xorwf    INDF0,f       ;      // w == contents for inserted cell for fsr0
xorwf    INDF0,w       ;      // *fsr0 == contents to overwrite in fsr0
xorwf    INDF0,f       ;      swap(&w, fsr0);

xorwf    temp,f        ;      // w == contents just overwritten in fsr0
xorwf    temp,w        ;      // temp == contents for inserted cell (fsr1)
xorwf    temp,f        ;      swap(&w, &temp);

xorwf    INDF1,f       ;      // w == contents for inserted cell in fsr1
xorwf    INDF1,w       ;      // *fsr1 == contents to overwrite in fsr1
xorwf    INDF1,f       ;      swap(&w, fsr1);

xorwf    temp,f        ;      // w == contents just overwritten in fsr1
xorwf    temp,w        ;      // temp == contents just overwritten in fsr0
xorwf    temp,f        ;      swap(&w, &temp);

addfsr   FSR0,+1       ;      // w == contents just overwritten in fsr0
addfsr   FSR1,+1       ;      // temp = contents just overwritten in fsr1

movf     INDF0,f       ;
btfss    STATUS,Z     ;
bra      groloop      ;      }

movwi    0[FSR0]       ;      // append the final overwritten contents
movf     temp,w        ;      *fsr0 = w; // this will be maxnon0 for last
movwi    0[FSR1]       ;      *fsr1 = w = temp;
movf     allocated,w   ;      w = allocated;
bra      done          ;      goto done; // return the fsr0 address added

free
movf     zOS_MSK,w     ;      //////////////////////////////////////
andlw    fi            ;      ////////////////////////////////// free() //////////////////////////////////
btfsc    STATUS,Z     ;
bra      invalid      ;      } else if (zOS_MSK & fi)

floop
zOS_LOC  FSR0,BSR,adrrary

moviw    FSR0++        ;      for (fsr0 = (bsr<<7) + adrrary;
xorwf    zOS_AR0,w     ;      fsr0 < adrrary + tblrows;//FIXME:sorted!
btfsc    STATUS,Z     ;      fsr0++) //could quit early!
bra      ffound        ;

movlw    adrrary+tblrows ;
xorwf    FSR0L,w       ;
andlw    0x7f          ;
btfss    STATUS,Z     ;
bra      floop         ;

ffound
bra      invalid      ;      if (*fsr0 == zOS_AR0) {
if (tblrows & 0x20)
addfsr   FSR0,0x1f     ;
addfsr   FSR0,tblrows-0x1f;

else
addfsr   FSR0,tblrows ;      fsr0 = sizarray + (fsr0 - adrrary);
endif
moviw    --FSR0        ;      w = *--fsr0;
clr      INDF0         ;      *fsr0 = 0;
bra      done          ;      }

invalid
clr      w             ;      else invalid: w = 0; // can't malloc nor free
done
zOS_RFS  WREG          ;      done: return w;

task
local    iniarray,coalesc,coaloop,coscoot

zOS_DIS  GIE,0
zOS_LOC  FSR0,BSR,0x70

iniarray
clr      w             ;      task: INTCON &= ~(1<<GIE);
movwi    --FSR0        ;      for (fsr0 = (bsr<<7)|(adrrary+tblsize);
movlw    adrrary        ;      fsr > adrrary; fsr--)
xorwf    FSR0L,w        ;      *fsr = 0; // zero each address and size entry
andlw    0x7f          ;
btfss    STATUS,Z     ;
bra      iniarray      ;

zOS_MY2  FSR1

movlw    membase        ;      // except first address entry is start of heap
movwi    0[FSR1]        ;      (0x70|(bsr<<1))[0] =
movwi    0[FSR0]        ;      adrrary[0] = membase; // first allocatable
movlw    membase+memsize ;      // and second address entry is the end of heap
movwi    1[FSR1]        ;      (0x70|(bsr<<1))[1] =
movwi    1[FSR0]        ;      adrrary[1] = membase+memsize;//max allocatable
zOS_ENA

coalesc
zOS_SWI  zOS_YLD
zOS_LOC  FSR0,BSR,adrrary+1
zOS_LOC  FSR1,BSR,sizarray

coaloop
moviw    ++FSR0         ;      do { // combine adjacent rows whose size are 0
btfsc    STATUS,Z       ;      zOS_SWI(zOS_YLD); // only 1 pass per schedule
bra      coalesc        ;      for (fsr0 = &adrrary[1], fsr1 = &sizarray[0];
moviw    FSR1++         ;      *++fsr0;
btfss    STATUS,Z       ;      fsr1++)
bra      coaloop        ;      if (0[fsr1] == 0 && 1[fsr1] == 0) {
moviw    0[FSR1]        ;      // fsr1->redundant row siz, trails fsr0->adr
btfss    STATUS,Z       ;      do {
bra      coaloop        ;      uint8_t w = *++fsr1;

coscoot
moviw    ++FSR1         ;      -1[fsr1] = w;
movwi    -1[FSR1]       ;      w = *fsr0++;
moviw    FSR0++         ;      } while ((-2[fsr0] = w) != 0);
movwi    -2[FSR0]       ;      break;
btfss    STATUS,Z       ;      }
bra      coscoot        ;      } while (1);
bra      coalesc        ;      decl:

decl
zOS_ADR  task,zOS_UNP   ;      fsr0 = task & 0x7fff;// MSB 0 => unprivileged
movlw    low isr        ;      w = zOS_ARG(0, isr & 0x00ff);
zOS_ARG  0
movlw    high isr       ;      w = zOS_ARG(1, isr>>8);
zOS_ARG  1
movlw    0              ;      w = zOS_ARG(2, 0); // no hardware interrupts
zOS_ARG  2
movlb    0              ;      // still in job "0": don't forget this!!!!

#if 0

```

```

        movlw    mi|fi          ; w = zOS_ARG(3, mi/*malloc()*/ | fi/*free()*/);
        zOS_ARG 3
        zOS_LAU FSR0
#endif

        endm                ;} // zOS_HEA()

;;; simple output-only console job with circular buffer
zOS_HEX macro
        andlw    0x0f          ;
        addlw    0x06          ;
        btfsc    WREG,4        ;inline char zOS_HEX(uint8_t w) {
        addlw    0x07          ; return (w & 0x0f > 9) ? '0'+w : 'A'+w-10;
        addlw    0x2a          ;} // zOS_HEX()
        endm

zOS_IHF macro    ofs,fsrsrc,fsrdst
        local    src,dst
        if (fsrsrc & 3)
src set 1
        else
src set 0
        endif
        if (fsrdst & 3)
dst set 1
        else
dst set 0
        endif

        moviw    ofs[FSR#v(src)] ;inline void zOS_IHF(int8_t ofs, int fsrnum,
        swapf    WREG,w          ; char* file) {
        zOS_HEX
        movwi    FSR#v(dst)++    ; file[0] = zOS_HEX(ofs[fsrnum] >> 4);
        moviw    ofs[FSR#v(src)] ; file[1] = zOS_HEX(ofs[fsrnum]);
        zOS_HEX
        movwi    FSR#v(dst)++    ;} // zOS_IHF()
        endm

zOS_UNW macro    job                ;inline void zOS_UNW(int8_t job) { }
        zOS_MEM FSR0,job,zOS_PCH; fsr0 = 0x10 * (1 + job) + zOS_PCH;
        bcf     INDF0,zOS_WAI      ; *fsr0 &= ~(1 << zOS_WAI); // now runnable
        endm                ;} // zOS_UNW()

zOS_OUT macro    swinum,str,temp
        local    agent,pre,post,setup,len,sloop,loop
        bra      setup            ;inline void zOS_OUT(uint8_t swinum, char* str,
agent                ;
        brw                ; uint8_t* temp) { // no '\0'
pre
        dt      str
post
        len    set    post-pre
        if (len > 254)
                error "string too long"
        endif

        if (len)
setup
        movlw    len                ; zOS_SWI(zOS_YLD); // get buffer empty as poss.
        movwf    temp                ; for (*temp = strlen(str); *temp; --*temp) {
sloop
        zOS_SWI zOS_YLD
loop
        movf     temp,w                ; zOS_ARG(0, w = str[strlen(str) - *temp]);
        sublw    len                ; while (zOS_SWI(swinum) != 1) { // buffer full
        pagesel agent
        call    agent                ; zOS_SWI(zOS_YLD); // flush buffer, retry
        zOS_ARG 0

```

```

        else
sloop
        zOS_SWI zOS_YLD
setup
        if (temp - zOS_AR0)
        if (temp - WREG)
                movf temp,w          ;
        endif
        zOS_ARG 0
        endif

        zOS_SWI swinum
        decfsz    WREG                ; zOS_ARG(0, w = str[strlen(str) - *temp]);
        bra      sloop                ; }

        if (len)
        decfsz    temp,f                ; }
        bra      loop                ;} // zOS_OUT()
        endif
        endm

zOS_PSH macro    reg
        movf     zOS_ME                ;inline void zOS_PSH(uint8_t* reg) {
        ;; bcf    INTCON,GIE
        banksel    TOSH
        incf     STKPTR,f                ; STKPTR++; // caller should've masked interrupts
        movwf    TOSH                ; TOSH = bsr; // must store bsr so we can go back
        if (reg-BSR)
        movf     reg,w                ; if (reg != &bsr)
        movwf    TOSL                ; TOSL = *reg;
        movf     TOSH,w                ; bsr = TOSH;
        endif
        movwf    BSR                ;} // zOS_PSH()
        ;; bsf    INTCON,GIE
        endm

zOS_POP macro    reg
        ;; bcf    INTCON,GIE
        banksel    STKPTR
        if (reg-BSR)
        movf     TOSL,w                ;inline void zOS_POP(uint8_t* reg) {
        movwf    reg                ; if (reg != &bsr) *reg = TOSL;
        endif
        movf     TOSH,w                ; bsr = TOSH;
        decf     STKPTR,f                ; STKPTR--; // caller should've masked interrupts
        movwf    BSR                ;} // zOS_POP()
        ;; bsf    INTCON,GIE
        endm

zOS_RDF macro
#ifdef EEADRL
        zOS_ADL equ    EEADRL
        zOS_ADH equ    EEARH
        zOS_RDL equ    EEDATL
        zOS_RDH equ    EEDATH
        banksel    EECON1
        bcf     EECON1,CFG5                ;inline void zOS_RDF(void) { // for EEAR micro
        bsf     EECON1,EEPGRD                ; EECON1 &= ~(1<<CFG5);
        bsf     EECON1,RD                ; EECON1 |= 1<<EEPGRD;
        nop                ; EECON1 |= 1<<RD;
        nop                ;} // zOS_RDF()
#else
#ifdef PMADRL
        zOS_ADL equ    PMADRL
        zOS_ADH equ    PMADRH
        zOS_RDL equ    PMDATL
        zOS_RDH equ    PMDATH
        banksel    PMCON1

```

```

        bcf     PMCON1,CFGFS ;inline void zOS_RDF(void) { // for PMADR micros
        bsf     PMCON1,RD    ; PMCON1 &= ~(1<<CFGFS);
        nop     ; PMCON1 |= 1<<RD;
        nop     ;} // zOS_RDF()
#else
#ifdef NVMDARL
zOS_ADL equ NVMDARL
zOS_ADH equ NVMDARH
zOS_RDL equ NVMDATL
zOS_RDH equ NVMDATH
banksel NVMCON1
        bcf     NVMCON1,NVMREGS ;inline void zOS_RDF(void) { // for NVM micros
        bsf     NVMCON1,RD    ; NVMCON1 &= ~(1<<CFGFS); NVMCON1 |= 1<<RD;
#endif
#endif
#endif

        endm ;} // zOS_RDF()

zOS_STR macro swinum
        local loop,done
        bcf     INTCON,GIE ;inline void zOS_STR(const char* fsr0,
zOS_PSH BSR
        banksel zOS_ADL
        movf    FSR0L,w ; uint8_t swinum) {
        movwf   zOS_ADL ; INTCON &= ~(1<<GIE);
        movf    FSR0H,w ; zOS_PSH(&bsr); // need a bank change for reads
        movwf   zOS_ADH ; for (zOS_AD = fsr0; *zOS_AD; zOS_AD++) {

loop
        zOS_RDF
        rlf     zOS_RDL,w ; zOS_RDF(); // read packed 14-bit contents
        rlf     zOS_RDH,w ;
        btfscc STATUS,Z ;
        bra     done ; if ((w = (zOS_RDH<<1)|(zOS_RDL>>7)) != '\0'){
        movwf   zOS_AR0 ; zOS_ARG(0, w);
        zOS_POP BSR
        zOS_OUT swinum,"",zOS_AR0
        bcf     INTCON,GIE ; zOS_POP(&bsr); // back to the expected bank
        zOS_PSH BSR
        banksel zOS_RDL
        movf    zOS_RDL,w ; zOS_OUT(swinum,"",zOS_AR0); // print ASCII
        andlw   0x7f ; INTCON &= ~(1<<GIE); // undo SWI GIE toggle
        btfscc STATUS,Z ; zOS_PSH(&bsr);
        bra     done ; if ((w = zOS_RDL & 0x7f) != '\0') {
        movwf   zOS_AR0 ; zOS_ARG(0, w);
        zOS_POP BSR
        zOS_OUT swinum,"",zOS_AR0
        bcf     INTCON,GIE ; zOS_POP(&bsr); // back to the expected bank
        zOS_PSH BSR
        banksel zOS_ADL
        incfsz  zOS_ADL,f ; zOS_SWI(swinum,"",zOS_AR0); // print ASCII
        bra     loop ; INTCON &= ~(1<<GIE); // undo SWI GIE toggle
        incf    zOS_ADH,f ; zOS_PSH(&bsr);
        bra     loop ; } else break;

done
        zOS_POP BSR ; } else break;
        bsf     INTCON,GIE ; } zOS_POP(&bsr); INTCON |= 1<<GIE;
        endm ;} // zOS_STR()

zOS_PUT macro fsrnum,max,wrap,p
        local fsrn
        if (fsrnum & 3)
        fsrn set 1
        else
        fsrn set 0
        endif
        movwi   FSR#v(fsrn)++ ;inline int8_t zOS_PUT(char**fsrnum,uint7_t max,
        movf    FSR#v(fsrn)L,w ; char* wrap, char* p, char w) {
        andlw   0x7f ; *(&fsrnum)++ = w;
        xorlw   max ; // w gets put in buffer regardless, but caller

```

```

        swapf   wrap,w ; // only updates the local pointer if not full
        btfscc STATUS,Z ; // (i.e. Z not set) by xor return value with p
        swapf   FSR#v(fsrn)L,w ; *fsrnum = (*fsrnum&0x7f==max) ? wrap : *fsrnum;
        swapf   WREG ; return (*fsrnum & 0x00ff) ^ p; //0 if full, or
        movwf   FSR#v(fsrn)L ; // new pointer value xor p if not
        xorwf   p,w ;} // zOS_PUT()
        endm

zOS_BUF macro fsrnum,max,ptr
        local ascii,err1,done
        local fsrn
        if (fsrnum & 3)
        fsrn set 1
        else
        fsrn set 0
        endif
        lsr     zOS_ME ;inline int8_t zOS_BUF(char**fsrnum,uint7_t max,
        movwf   FSR#v(fsrn)H ; char** ptr, char w) { // p0, p1, wrap
        movf    1+ptr,w ; // must be in job bank already, interrupts off
        movwf   FSR#v(fsrn)L ; fsr0 = (bsr<<7) | ptr[1]; // insertion pointer

        movf    zOS_AR0,w ; if ((w = zOS_AR0) == 0) { // 2-digit hex byte
        btfscc STATUS,Z ; w = zOS_HEX(zOS_AR1>>4); // convert high nyb
        bra     ascii ; w = zOS_PUT(fsrnum, max, ptr[0], w); // room?

        swapf   zOS_AR1,w ; if (w == 0)
        zOS_HEX
        zOS_PUT fsrnum,max,2+ptr,ptr
        btfscc STATUS,Z ; return 0; // buffer was full
        bra     done ; ptr[1] = w^ptr[0]; // correctly updated
        xorwf   ptr,w ; w = zOS_HEX(zOS_AR1); // convert low nybble
        movwf   1+ptr ; w = zOS_PUT(fsrnum, max, ptr[0], w); // room?

        movf    zOS_AR1,w ; if (w == 0)
        zOS_HEX
        zOS_PUT fsrnum,max,2+ptr,ptr
        btfscc STATUS,Z ; return 1; // buffer filled after first char
        bra     err1 ; ptr[1] = w^ptr[0]; // correctly updated
        xorwf   ptr,w ; w = 2;
        movwf   1+ptr ; } else { // print an ascii character
        movlw   2 ; if ((w = zOS_PUT(fsrnum,max,ptr[0],w)) == 0)
        bra     done ; return 0; // buffer was full

        ascii
        zOS_PUT fsrnum,max,2+ptr,ptr
        btfscc STATUS,Z ; ptr[1] = w^ptr[0]; // correctly updated
        bra     done ; w = 1;
        xorwf   ptr,w ; }
        movwf   1+ptr ; return w; // num of characters added to buffer

        err1
        movlw   1 ;} // zOS_BUF()

        done
        endm

zOS_NUL macro hwflag
        bra     decl ; goto decl;
        local task,isr,decl ; task: do {

        task
        zOS_SWI zOS_YLD ; zOS_SWI(zOS_YLD);
        bra     task ; } while (1);

        isr
        banksel zOS_T0F ; isr:
        bcf     zOS_T0F,T0IF ; zOS_T0F &= ~(1<<T0IF); // clear interrupt flag
        zOS_RFI ; zOS_RFI(); // and go back to scheduler

        decl
        zOS_ADR task,zOS_UNP ; fsr0 = task & 0x7fff; // MSB 0 => unprivileged
        movlw   low isr ; w = zOS_ARG(0, isr & 0x00ff);
        zOS_ARG 0

```

```

        movlw    high isr      ; w = zOS_ARG(1, isr>>8);
        zOS_ARG 1              ; w = zOS_ARG(2, 1<<T0IF);
        movlw    hwflag       ; w = zOS_ARG(3, 0 /* no SWI */);
        zOS_ARG 2              ;
        clrw                     ;
        zOS_ARG 3              ;
        movlb    0              ; // still in job "0": don't forget this!!!!
        endm

zOS_CON macro    p,rat,rts,hb,pin;inline void zOS_CON(int8_t p,int8_t rat,int8_t
local            task,isr,loop,decl
bra              decl          ;          rts,int8_t* hb,int8_t pin){

        ;; initialize constants and variables
local    t0div,t0rst

t0div    set 0
t0rst    set 1

        local    p0,p1,wrap,t0scale,isradrl,isradrh,tskadrl,tskadrh,optadrl
local    optadrh,accumul,accumuh,numbase,destreg,destreh,char_io,buf,max

        ;; 0x20~24 reserved for zOS_CON
p0        set    0x20
p1        set    0x21
wrap      set    0x22
t0scale   set    0x23

        ;; 0x24~28 reserved for zOS_INP
isradrl   set    0x24
isradrh   set    0x25
tskadrl   set    0x26
tskadrh   set    0x27

        ;; 0x28~2F reserved for zOS_MON and derivations e.g. zOS_MAN
optadrl   set    0x28
optadrh   set    0x29
accumul   set    0x2a
accumuh   set    0x2b
numbase   set    0x2c
destreg   set    0x2d
destreh   set    0x2e
char_io   set    0x2f
buf       set    0x30
max       set    0x70

;copy the preceding lines rather than including this file, as definitions for
;zOS_MON()-derived macros referring to these local variables wouldn't open it
;until expansion and would throw an undefined-var error during the processing

        local    uatbase,uatxmit
        if (p == 0)
uatbase    set    TXREG & 0xff80
uatxmit    set    TXREG & 0x001f ; mask off just the SFR space
rtsflag    set    TXIF
        else
uatbase    set    TX#v(p)REG & 0xff80
uatxmit    set    TX#v(p)REG & 0x001f ; mask off just the sfr SFR
rtsflag    set    TX#v(p)IF
        endif
task

        zOS_DIS GIE,0          ; goto decl;
        movlw    high uatbase   ;// all init that requires knowledge of BSR here
        movwf    FSR0H          ;task:
        zOS_MY2 FSR0            ; zOS_DIS(&fsr0, zOS_JOB); // interrupts off!
        movlw    0xff           ; fsr0 = 0x70 + (bsr<<1); //global always visible
        movwi    t0div[FSR0]    ; 0[fsr0] = 0xff; // live TMR0 postscaler divider
        movlw    0x00           ;
        movwi    t0rst[FSR0]    ; 1[fsr0] = 0x00; // live reset value for TMR0
        movlw    low uatbase    ;

```

```

        movwf    FSR0L          ; const int8_t* uatbase = uatxmit & 0xff80;
        rrf      zOS_ME          ; fsr0 = uatbase;
        clrw                     ; const char* max = 0x70;
        rrf      WREG            ; static char *p0, *p1, buf[]; //p0:task, p1:ISR
        iorlw    buf            ;
        movwf    wrap           ; const char* wrap = ((bsr&1)<<7) | buf;
        movwf    p0             ;
        movwf    p1             ; p0 = p1 = wrap; // reset value if they max out
        zOS_ENA                ; zOS_ENA(); // interrupts on after init done

```

```
zOS_OUT 0xff,"\\r\\nWelcome to zOS\\r\\n",char_io
```

loop

```

zOS_SWI zOS_YLD          ; do {
        movlw    high rts      ; zOS_YLD();
        movwf    FSR1H          ;
        movlw    low rts       ; // wait for SWI to store char(s) in buf[]
        movwf    FSR1L          ;
        btfs     INDF1,rtsflag   ; if (*(fsr1 = rts) & (1<<rtsflag) == 0) //full
        bra      loop          ; continue; // yield (still sending or no char)
        lsrwf    zOS_ME          ;
        movwf    FSR1H          ;
        movf     p0,w           ; // READY TO SEND, AND...
        movwf    FSR1L          ;
        xorwf    p1,w           ; fsr1 = (bsr<<7) | p0;
        btfs     STATUS,Z        ; if (p0 == p1)
        bra      loop          ; continue; // nothing to do
        moviw    FSR1++         ;
        movwi    uatxmit[FSR0]   ; uatxmit[fsr0] = *fsr1++; // send a character
        movf     FSR1L,w         ;
        movwf    p0             ; p0 = fsr1 & 0x00ff; // wrap around to buf+0
        andlw    0x7f           ;
        xorlw    max            ;
        btfs     STATUS,Z        ;
        bra      loop          ; if (p0 & 0x7f == max) // ignore low bank bit
        movf     wrap,w         ; p0 = wrap; // =buf xor the lowest bank bit
        movwf    p0             ;
        bra      loop          ; } while (1);

```

```

;; HWI will be coming from a tmr0 expiration, for the blinking heartbeat
;;
;; SWI will be coming from a job that wants to send a character
;; in which case the ISR stores it, advancing p1 and returning the
;; number of characters stored in the buffer
;; Note: caller needs to make sure to check status of return value for
;; != 0, just in case job is in between sleeps or with a full buffer

```

isr

```

local    done,do_swi,nottmr

;; if it's a simple and frequent timer overflow interrupt finish quickly
banksel  zOS_T0F
        btfs     zOS_T0F,T0IF    ; if (/*presumed true:(zOS_T0E & (1<<T0IE)) &&*/
        bra      nottmr         ; (zOS_T0F & (1<<T0IF))) { // timer overflow
        bcf      zOS_T0F,T0IF    ; zOS_T0F &= ~(1<<T0IF); // clear interrupt flag

        ;; get fsr0 pointing to tmr0 postscaler/reset value
        movf     zOS_JOB,w        ;isr:
        movwf    BSR              ; bsr = zos_job;
        zOS_MY2 FSR0L            ; fsr0 = 0x70 | (bsr < 1);

        ;; with fsr0 pointing to global pair, point fsr1 to local mem("t0scale")
        zOS_LOC FSR1,zOS_JOB,t0scale
        banksel  TMR0
        moviw    t0rst[FSR0]      ; fsr1 = (zOS_JOB << 7) | t0scale;
        btfs     WREG,7           ; bsr = TMR0 >> 7; //now invalid for this branch
        movwf    TMR0            ; if (t0rst[fsr0] < 128) // max 7 bit TMR0 reset
        decfsz   INDF1,f         ; TMR0 = t0rst[fsr0]; // or chance of deadlock
        bra      done            ; if (--*fsr1 == 0) {

```

```

bankssel hb
movf    INDF0,w      ;
btfsc   STATUS,Z    ;
movlw   1            ; if (*fsr0 == 0) // disallow zero postscaler
movwf   INDF0        ; *fsr0 = 1;
movwf   INDF1        ; *fsr1 /*countdown*/ = *fsr0 /*postscaler*/;
movlw   (1<<pin)     ;
xorwf   hb,f         ; hb ^= 1 << pin;
bra     done         ; } else {

;; check for validated SWI first since it will be in zOS_MSK, else a HWI
nottmr
movf    zOS_MSK,f    ; if (zOS_MSK) { // a SWI to buffer a character
btfss   STATUS,Z     ; w = zOS_BUF(&fsr0, max, p0); // zOS_AR0,_AR1
bra     do_swi       ; zOS_RFS(w); } else zOS_RET(); // not ours(!)
zOS_RET

;; point fsr0 to uatbase (again?), point fsr1 to p0
do_swi
movf    zOS_JOB,w    ;
movwf   BSR          ;
zOS_BUF FSR0,max,p0  ; }
zOS_RFS WREG         ; zOS_RFI(); // HWI finished
done
zOS_RFI              ;}

;; intialize the UART peripheral, job handle and first three arguments
decl
bankssel uatbase
bcf     RCSTA,SPEN    ;decl: // all init that is BSR independent here
#if 1
bcf     RCSTA,CREN    ; RCSTA &= ~(1<<SPEN)|(1<<CREN));
#endif
bcf     TXSTA,TXEN    ; TXSTA &= ~(1<<TXEN);
local  brgval,brgvalm,brgvalh,brgvall
#ifdef BRG16
brgval  set    rat>>2
brgvalm set    brgval-1
brgvalh set    high brgvalm
brgvall set    low brgvalm
bankssel uatbase
bsf     BAUDCON,BRG16 ; // section 26.1.2.8 of 16F1847 steps below:
bankssel uatbase
bcf     TXSTA,SYNC    ; // (1) "Initialize..the desired baud rate"
bsf     TXSTA,BRGH    ; BAUDCON |= 1<<BRG16; // 16-bit generator
movlw   brgvall       ; TXSTA &= ~(1<<SYNC); // async mode
movwf   SPBRGL        ; TXSTA |= 1<<BRGH; // high speed
movlw   brgvalh       ;
movwf   SPBRGH        ; SPBRG = (rat/4) - 1;
bcf     BAUDCON,SCKP  ; BAUDCON &= ~(1<<SCKP); // "SCKP..if inverted"
#else
brgval  set    rat>>4
brgvalm set    brgval-1
brgvalh set    0
brgvall set    low brgvalm
bsf     TXSTA,BRGH    ; TXSTA |= 1<<BRGH; // (1) the desired baud rate
bankssel uatbase
movlw   brgvall       ;
movwf   SPBRG         ; SPBRG = (rat/16) - 1;
#endif
#if 1
bankssel uatbase
bsf     RCSTA,SPEN    ; // (3) "Enable..by setting..SPEN"
bcf     RCSTA,RX9     ; RCSTA &= ~(1<<RX9); // (5) "9-bit..set..RX9"
bsf     RCSTA,CREN    ; RCSTA |= (1<<SPEN) | (1<<CREN); // (6) "CREN"
#endif
bankssel uatbase
bsf     TXSTA,TXEN    ; TXSTA |= 1<<TXEN; // (5) "Enable..by..TXEN"
#if 1

```

```

bankssel PIE1
bsf     PIE1,RCIE     ; PIE1 |= 1<<RCIE; //(4) "Set..RCIE..and..PEIE"
#endif

zOS_ADR task,zOS_UNP  ; fsr0 = task & 0x7fff; // MSB 0 => unprivileged
movlw   low isr       ; w = zOS_ARG(0, isr & 0x00ff);
zOS_ARG 0
movlw   high isr      ; w = zOS_ARG(1, isr>>8);
zOS_ARG 1             ; w = zOS_ARG(2, (0<<TXIF)|(1<<T0IF));
movlw   (0<<TXIF)|(1<<T0IF)
zOS_ARG 2
movlb   0             ; // still in job "0": don't forget this!!!!
endm                ;} // zOS_CON()

;; remnants of an early experiment to allow bank changing outside ISR
;; to read SFR's is now deprecated, only known use is in olirelay.asm
zOS_R macro file,bankf,prsrv;inline int8_t zOS_R(const int8_t* file, int8_t bank, int8_t prsrv) {
if (prsrv)
movf    INTCON,w
bcf     INTCON,GIE
movwf   zOS_AR1
else
bcf     INTCON,GIE
endif
if file & 0x60
error "tried to access disallowed RAM range (global or another job's)"
endif
bankssel file        ; INTCON &= ~(1<<GIE); // access zOS_AR* globals
movf    file,w        ; bsr = file >> 7;
movwf   zOS_AR0       ; zOS_AR0 = *file; // any 0-0x1f SFR in any bank
movf    bankf,w       ; bsr = bankf;
movwf   BSR           ; w = zOS_AR0;
movf    zOS_AR0,w     ; if (prsrv && (zOS_AR1 & (1<<GIE)))
if prsrv
btfss   zOS_AR1,GIE   ; INTCON |= 1<<GIE; // restore interrupt state
endif
bsf     INTCON,GIE    ; return w;
endm                ;} // zOS_R()

;;; like zOS_CON, but also accepts console input for command-line interaction
zOS_INP macro p,ra,rt,h,pi,isr;inline void zOS_INP(int8_t p, int8_t ra, int8_t
local rxtask,no_opt,rxisr,rxdecl
bra     rxdecl        ; rt, int8_t* h, int8_t pi, void(*isr)() {

;; reserve constants and variables
local p0,p1,wrap,t0scale,isradrl,isradrh,tskadrl,tskadrh,optadrl
local optadrh,accumul,accumuh,numbase,destreg,destreh,char_io,buf,max

;; 0x20~24 reserved for zOS_CON
p0      set    0x20
p1      set    0x21
wrap    set    0x22
t0scale set    0x23

;; 0x24~28 reserved for zOS_INP
isradrl set    0x24
isradrh set    0x25
tskadrl set    0x26
tskadrh set    0x27

;; 0x28~2F reserved for zOS_MON and derivations e.g. zOS_MAN
optadrl set    0x28
optadrh set    0x29
accumul set    0x2a
accumuh set    0x2b
numbase set    0x2c
destreg set    0x2d
destreh set    0x2e
char_io set    0x2f

```



```
buf      set      0x30
max      set      0x70
```

```
;copy the preceding lines rather than including this file, as definitions for
;zOS_MON()-derived macros referring to these local variables wouldn't open it
;until expansion and would throw an undefined-var error during the processing
```

```
        local      uarbase,uarecv,rxflag
        if (p == 0)
uarbase set      RCREG & 0xff80
uarrecv set      RCREG & 0x7f
rxflag  set      RCIF
        else
uarbase set      RC#v(p)REG & 0xff80
uarrecv set      RC#v(p)REG & 0x7f
rxflag  set      RC#v(p)IF
        endif
```

```
;;; FIXME: haven't actually written the var init code for zOS_MON et al yet
```

```
rxtask  movf      zOS_JOB,w          ; goto rxdecl;
        movwf     BSR                ;rxtask:
        movf      optadrl,w          ;
        movwf     PCLATH             ;
        iorwf     optadrl,w          ;
        btfsc     STATUS,Z           ;
        bra       no_opt             ;
        movf      optadrl,w          ; if ((optadrl<<8) | optadrl)
        callw     ; (* (optadrl<<8) | optadrl) (); //returns to:
;;; FIXME: do anything interesting with return value? 0 sent if nothing happened
no_opt  movf      tskadrl,w           ;
        movwf     PCLATH             ; goto (tskadrl<<8) | tskadrl; // zOS_CON() code
        movf      tskadrl,w           ;
        movwf     PCL                ;callw ;
```

```
rxisr   movf      zOS_JOB,w          ;rxisr:
        movwf     BSR                ; bsr = zOS_JOB; // isr starts with unknown bank
#if 0
        movlw     low uarbase         ;
        movwf     FSR0L               ;
        movlw     high uarbase        ;
        movwf     FSR0H               ; fsr0 = uarbase;
        zOS_LOC  FSR1,zOS_JOB,buf,p0 ;
#endif
```

```
        movf      isradrl,w          ;
        movwf     PCLATH             ;
        movf      isradrl,w          ; if (rt && (1<<RCIF) == 0) // SWI, not inp char
        banksel   uarbase            ;
        btfss     rt,rxflag           ; goto (isradrl<<8)|isradrl; //zOS_CON takes SWI
        movwf     PCL                 ; else {
        bcf       rt,rxflag           ; rt &= ~(1<<RCIF);
#ifdef CAUTIOUS
        btfss     RCSTA,OERR          ;
        bra       noovrrn             ; if ((uarbase | RCSTA) & (1<<OERR)) {
        movlw     '1'                 ; zOS_AR0 = '1';
        movwf     zOS_AR0             ; zOS_BUF(zOS_JOB, p0);
        zOS_BUF  zOS_JOB,buf,p0      ; }
noovrrn
#endif
```

```
        movf      RCREG,w             ; // this read removes it from the FIFO
        movwf     zOS_AR0             ; zOS_AR0 = RCREG;
```

```
#ifdef CAUTIOUS
        btfss     RCSTA,OERR          ; if (RCSTA & (1<<OERR)) // rx overrun
        bcf       RCSTA,CREN          ; RCSTA &= ~(1<<CREN); // cleared by disable
        bsf       RCSTA,CREN          ; RCSTA |= 1<<CREN; // (re-)enable reception
#endif
```

```
pagesel isr          ; if (zOS_AR0)
btfss     STATUS,Z    ; goto isr; // continue with parser
goto      isr         ; zOS_RFI(); //return from interrupt
zOS_RFI    ; }
```

```
rxdecl
```

```
        local      vars,arg0,arg1,adrl,adrh,optl,opth,chio
vars     set      0x20
arg0     set      isradrl-vars
arg1     set      isradrlh-vars
adrl     set      tskadrl-vars
adrh     set      tskadrlh-vars
optl     set      optadrl-vars
opth     set      optadrlh-vars
chio     set      char_io-vars
```

```
zOS_CON p,rat,rts,hb,pin
zOS_LAU FSR1H
zOS_LOC FSR1L,FSR1H,vars
movf     zOS_AR0,w          ;rxdecl:
movwi    arg0[FSR1]         ; zOS_CON(p,rat,rts,hb,pin); // extend zOS_CON()
movf     zOS_AR1,w          ; zOS_LAU(&fsr1); // by rewriting after launch
movwi    arg1[FSR1]         ; fsrl <= 7;
movf     FSR0L,w            ; isradrl[fsr1] = (zOS_AR1<<8) | zOS_AR0;
movwi    adrl[FSR1]         ;
movf     FSR0H,w            ;
movwi    adrl[FSR1]         ; tskadrl = fsr0; // all non-rx tasks here
movlw    0                  ;
movwi    chio[FSR1]         ; char_io[fsr1] = 0; // nonzero = action to take
addfswr  FSR1,optl          ; fsrl += optadrl; // caller sets optional task
movwi    0[FSR1]            ;
movwi    1[FSR1]            ; optadrl[fsr1] = ((*void)()) 0; // no func
rlf      FSR1L,w            ; w = fsr1 >> 7; // restore zOS_LAU() job number
rlf      FSR1H,w            ;
zOS_MEM  FSR0,WREG,0
movlw    low rxtask         ; fsr0 = 0x10 + w << 4;
movwi    zOS_HDL[FSR0]      ;
movwi    zOS_PCL[FSR0]      ;
movlw    high rxtask        ;
movwi    zOS_PCH[FSR0]      ; zOS_PC[fsr0] = rxtask;
iorlw    0x80               ;
movwi    zOS_HDH[FSR0]      ; zOS_HD[fsr0] = rxtask | 0x8000;
addfswr  FSR0,zOS_ISR       ; fsr0 += zOS_ISR; // last 4 bytes of job record
movlw    low rxisr          ; *fsr0++ = rxisr & 0x00ff;
movwi    FSR0++             ;
movlw    high rxisr         ; *fsr0++ = rxisr >> 8;
movwi    FSR0++             ;
movf     zOS_AR2,w          ; *fsr0++ |= (1<<RCIF); // |(0<<TXIF)|(1<<T0IF));
iorlw    1<<rxflag          ; // still in job "0"; caller sets any SWI value
movwi    FSR0++             ; } // zOS_INP()
endm
```

```
zOS_ACC macro      valregs,basereg
        clrfs      valregs          ;inline uint8_t zOS_ACC(uint8_t* valregs,uint8_t
        clrfs      1+valregs        ; *basereg) { // w unclobbered
        clrfs      basereg          ; *valregs = 0;
        bsf        basereg,3        ; return *basereg = 10; // decimal by default
        bsf        basereg,1        ; } // zOS_ACC()
        endm
```

```
zOS_PCT macro      reg
        movlw      0x7e             ; // 0 <= reg <= 100
        andwf      reg,w            ; w = reg & 0x7e; // 0 <= w <= reg (even, trunc)
        lslf      reg,f             ;
        lslf      reg,f             ; uint16_t c = reg * 4; // 0 <= reg <= 400
        btfsc     STATUS,C          ; if (c > 0xff)
        iorlw      0x01             ; w |= 1;
        addwf      reg,f            ; c = reg += w;
```

```

btfsc STATUS,C ; if (c > 0xff)
iorlw 0x01 ; w |= 1;
rrf WREG ; // 0 <= (w&1)*256 + reg <= 500
rrf reg,f ; reg = ((w&1)*256 + reg)/2; // 0 <= reg <= 250
endm

zOS_MON macro p,ra,rt,h,pi,isr;inline void zOS_MON(int8_t p, int8_t ra, int8_t
local monisr,monchr1,monchr2,monchr3,mondump,mondest,monram,monchr4
local monchr5,monchr6,monchr7,monchr8,monchr9,monprmp,monlast,endmon

zOS_INP p,ra,rt,h,pi,monisr
pagesel endmon ; rt, int8_t* h, int8_t pi, void(*isr)() {
goto endmon ; zOS_INP(p,ra,rt,h,pi,monisr); } // isr may be 0

local p0,p1,wrap,t0scale,isradrl,isradrh,tskadrl,tskadrh,optadrl
local optadrh,accumul,accumuh,numbase,destreg,destreh,char_io,buf,max

;; 0x20~24 reserved for zOS_CON
p0 set 0x20
p1 set 0x21
wrap set 0x22
t0scale set 0x23

;; 0x24~28 reserved for zOS_INP
isradrl set 0x24
isradrh set 0x25
tskadrl set 0x26
tskadrh set 0x27

;; 0x28~2F reserved for zOS_MON and derivations e.g. zOS_MAN
optadrl set 0x28
optadrh set 0x29
accumul set 0x2a
accumuh set 0x2b
numbase set 0x2c
destreg set 0x2d
destreh set 0x2e
char_io set 0x2f
buf set 0x30
max set 0x70

```

;copy the preceding lines rather than including this file, as definitions for  
;zOS\_MON()-derived macros referring to these local variables wouldn't open it  
;until expansion and would throw an undefined-var error during the processing

```

monback
andlw 0x3f ;void monback(uint3_t job, uint8_t ptr, char w){
btfsc STATUS,Z ; if (w &= 0x3f) {
return ; // 63 \b's should be enough in a buffer of 64
movwf zOS_AR1 ;
movlw 0x08 ;
movwf zOS_AR0 ; zOS_AR0 = '\b';

```

```

monloop
zOS_BUF zOS_JOB,buf,p0
andlw 0x1 ; for (zOS_AR1 = w; zOS_AR1; zOS_AR1--) {
btfsc STATUS,Z ; if (zOS_BUF(job, ptr) == 0) // buff full
return ; return;
decfsz zOS_AR1,f ; }
bra monloop ; }
return ;} // monback()

```

```

monhex
movlw '0' ;void monhex(uint3_t job, uint8_t ptr) {
movwf zOS_AR0 ; extern uint8_t accumuh;
zOS_BUF zOS_JOB,buf,p0
andlw 0x1 ; zOS_AR0 = '0';
btfsc STATUS,Z ; if (zOS_BUF(job, ptr) == 0) // buf full
return ; return;

```

```

movlw 'x' ;
movwf zOS_AR0 ; zOS_AR0 = 'x';
zOS_BUF zOS_JOB,buf,p0
andlw 0x1 ; if (zOS_BUF(job, ptr) == 0) // buf full
btfsc STATUS,Z ; return;
return ; monlsb(job, ptr, w = accumuh); // not accumul
movf accumuh,w ;} // monhex()

```

```

monlsb
clrf zOS_AR0 ;void monlsb(uint3_t job, uint8_t ptr, char w) {
movwf zOS_AR1 ; zOS_AR0 = 0; zOS_AR1 = w; monbuf(job, ptr);
bra monbuf ;} // monlsb()

```

```

moncrlf
movlw '\r' ;void moncrlf(uint3_t job, uint8_t ptr, char w){
movwf zOS_AR0 ; zOS_AR0 = '\r';
zOS_BUF zOS_JOB,buf,p0 ; if (zOS_BUF(zos_job, ptr) < 1)
andlw 0x1 ; return 0;
btfsc STATUS,Z ;
return ; zOS_AR0 = '\n';

```

```

monlf
movlw '\n' ; return zOS_BUF(zos_job, ptr, w);
movwf zOS_AR0 ;} // moncrlf() monlf()

```

```

monbuf
zOS_BUF zOS_JOB,buf,p0 ;void monbuf(uint3_t job, uint8_t ptr, char w) {
return ; return zOS_BUF(job,ptr,w); } // 0/1/2 printed

```

```

monisr
pagesel monbuf ;void monisr(void) {
movlw 0xe0 ; // from zOS_INP isr with char zOS_AR0>0
addwf zOS_AR0,w ;
btfsc WREG,7 ; // refuse to echo unprintable characters
call monbuf ; if (zOS_AR0 > 31 && monbuf(zos_job,p0) > 0) {
andlw 0x1 ; // successful echo into circular buffer
btfsc STATUS,Z ;
bra monlast ;
movf zOS_JOB,w ;
movwf BSR ; bsr = zos_job;// to access char_io var et al

```

```

movf zOS_AR0,w ; // handle '~' before the tolower() conversion
xorlw '~' ;
btfsc STATUS,Z ;
bra monchr1 ; if (zOS_AR0 == '~') {
comf accumul,f ; accumul = ~accumul;
comf accumuh,w ;
movwf accumuh ;
movwf char_io ; char_io = accumuh = ~accumuh; // preserve
pagesel monhex ; monhex(zos_job, p0);
call monhex ; accumul = accumuh; // accumuh overwritten
movf accumul,w ; monlsb(zos_job, p0);
movwf accumuh ;
pagesel monlsb ; accumul = char_io; // accumuh now restored
call monlsb ; char_io = 0; // completely handled in ISR
movwf char_io,w ; zOS_RFI();
movwf accumuh ;
clrf char_io ; }
zOS_RFI

```

```

monchr1
btfsc zOS_AR0,6 ; if (zOS_AR0 & 0x40)
bcf zOS_AR0,5 ; zOS_AR0 &= 0xdf; // zOS_AR0=tolower(zOS_AR0)
movf zOS_AR0,w ;//FIXME: ' { | } ~ DEL mapped onto @ [ \ ] ^ _
movwf char_io ;
xorlw 0x08 ; switch (char_io = zOS_AR0) {
btfsc STATUS,Z ; case '\b':
bra monchr2 ;
movlw '\r' ;

```

```

        pagesel monbuf
        call    monbuf          ; monbuf(zos_job, p0, '\r');
        bra     monprmp         ; goto monprmp;

monchr2
        movf    char_io,w       ;
        xorlw   0x0a            ;
        btfss   STATUS,Z        ; case '\r':
        bra     monchr3         ; monbuf(zos_job, p0, '\n');// follows the \r
        pagesel monlf
        call    monlf           ;

        movf    destreg,w       ; // repeat \r's can set a whole range of
        movwf   FSR0L,w         ; // addresses to zero
        movf    1+destreg,w     ;
        movwf   FSR0H           ; fsr0 = destreg;
        iorwf   FSR0L,w         ;
        btfsc   STATUS,Z        ;
        bra     monprmp         ; if (fsr0) { // destreg was set by ' ' or =
        movf    accumul,w       ; if (fsr0 & 0x8000 == 0)
        btfss   FSR0H,7         ;
        movwi   FSR0++          ; *fsr0 = accumul & 0x00ff; // not in flash
        movf    FSR0L,w         ;
        movwf   destreg         ;
        movf    FSR0H,w         ; destreg++; // advances for next access
        movwf   1+destreg       ; }
        bra     monprmp         ; goto monprmp;

monchr3
        movf    char_io,w       ;
        xorlw   0x20            ;
        btfsc   STATUS,Z        ; case ' ':
        bra     mondump         ;
        movf    char_io,w       ;
        xorlw   '.'            ;
        btfsc   STATUS,Z        ; case '.':
        bra     mondump         ;
        movf    char_io,w       ;
        xorlw   '='            ;
        btfss   STATUS,Z        ; case '=':
        bra     monchr4         ;

mondump
        movf    accumul,w       ; // pressing ' ' or '.' or '=' should apply
        iorwf   accumuh,w       ; // to the recently incremented address from
        btfsc   STATUS,Z        ; // a previous operation (if any) or to an
        bra     mondest         ; // an address typed immediately before it
        movf    accumul,w       ;
        movwf   destreg         ;
        movf    accumuh,w       ; if (accumul) // typed a value before ' ' /=
        movwf   1+destreg       ; destreg = accumul; // otherwise no clobber

mondest
        movf    destreg,w       ;
        movwf   FSR0L           ;
        movf    1+destreg,w     ;
        movwf   FSR0H           ; fsr0 = destreg;

        btfsc   1+destreg,7     ; if (destreg & 0x8000) { // flash, not RAM
        bra     monram          ;
;;; FIXME: access upper byte in Flash instead of printing it as zero
        clrf    accumuh         ;
        pagesel monhex
        call    monhex          ; monhex(zos_job, p0, accumuh=0);// put 0x00
        movf    destreg,w       ;
        movwf   FSR0L           ;
        movf    1+destreg,w     ;
        movwf   FSR0H           ; fsr0 = destreg; // monhex() clobbered fsr0
        movi    FSR0++          ;

        movwf   accumuh         ;
        movf    FSR0L,w         ;
        movwf   destreg         ; accumuh = *fsr0++;
        movf    FSR0H,w         ; destreg = fsr0;
        movwf   1+destreg       ; monlsb(zos_job, p0, accumuh); // LSB
        pagesel monlsb
        call    monlsb         ; monCrLf(zos_job, p0); // \r\n
;;; FIXME: disassemble the instruction here once the upper 6 bits are available
        pagesel monCrLf
        call    monCrLf         ; goto monprmp;
        bra     monprmp         ; }

monram
        movi    FSR0++          ;
        movf    FSR0L,w         ;
        movwf   destreg         ;
        movf    FSR0H,w         ;
        movwf   1+destreg       ;
        movwf   accumuh         ; accumuh = *(destreg = fsr0++);
        pagesel monhex
        call    monhex         ; monhex(

        movf    char_io,w       ;
        xorlw   '.'            ; // then exits in the '.' case to just print
        pagesel monCrLf
        btfss   STATUS,Z        ; if (char_io == '.')
        goto    monCrLf         ; goto monCrLf;

        movf    char_io,w       ; // or follow by 3 backspaces in the ' ' case
        xorlw   '='            ; // to show that \r will result in a 0 write
        btfss   STATUS,Z        ;
        movlw   3               ;
        pagesel monback
        call    monback         ; monback(zos_job, p0, (char_io == '=')?0:3);
        clrf    char_io         ; char_io = 0;
        zOS_RFI                 ; break;

monchr4
        movf    char_io,w       ;
        xorlw   'X'            ;
        btfss   STATUS,Z        ; case 'X':
        bra     monchr5         ;
        movlw   0x10            ; numbase = 16;
        movwf   numbase         ; char_io = 0;
        clrf    char_io         ; break;
        zOS_RFI

monchr5
        movf    char_io,w       ;
        xorlw   '%'            ;
        btfss   STATUS,Z        ; case '%':
        bra     monchr6         ;
        movlw   0x9b            ;
        addwf   accumul,w       ;
        movlw   0x66            ;
        btfss   WREG,7          ; if (accumul > 102)
        movwf   accumul         ; accumul = 102;
        zOS_PCT accumul
        movwf   accumul         ; accumul = zOS_PCT(accumul);
        movwf   accumuh         ; accumuh = accumul;
        pagesel monhex
        call    monhex          ; monhex(zos_job, p0); print as e.g. 50%0x7d
        clrf    accumuh         ; accumuh = 0;
        clrf    char_io         ; char_io = 0;
        zOS_RFI                 ; break;

monchr6
        movlw   0-0x10          ; default:
        addwf   char_io,f       ;

```

```

        btfsc    char_io,7      ;
        bra     monchr9        ; if ((char_io -= ('0'&0xdf /*0x10*/)) >= 0) {
        addwf   char_io,w      ;
        btfsc   WREG,7         ; if (char_io > 0x10)
        bra     $+3            ;
        movlw   0xf9           ;
        addwf   char_io,f      ; char_io -= 0x07; // 0x41->0x31->0x2a... so
        movf    char_io,f      ; // now in range 0x00-0x09,
        btfss   STATUS,Z       ; // or :=0x0a,...,?=0x0f,
        bra     monchr7        ; // or A=0x2a,B=0x2b,...
        movf    accumul,w      ; // G=0x30,...,Z=0x43
        iorwf   accumul,w      ; if ((char_io == 0) &&
        btfss   STATUS,Z       ; (accumul == 0) && (accumuh == 0)) {
        bra     monchr7        ; numbase &= ~2; // digit(s) leading 0(s),
        bcf     numbase,1      ; char_io = 0;
        clrf    char_io        ; break; // just go into octal mode
        zOS_RFI

monchr7
        movlw   0x50           ;
        andwf   char_io,w      ;
        btfss   STATUS,Z       ; } else if ((char_io & 0x50 == 0) // 0-9,a-f
        bra     monchr8        ; && (numbase & 0x10)) { // base 16
        btfss   numbase,4      ;
        bra     monchr8        ;
        swapf   accumul,w      ;
        movlw   0xf0           ;
        andwf   accumul,w      ; accumul <= 4;
        swapf   accumul,w      ;
        andlw   0x0f           ;
        iorwf   accumul,w      ; accumul |= accumul >> 4;
        movlw   0x0f           ;
        andwf   char_io,f      ; char_io &= 0x0f;
        andwf   accumul,f      ; accumul &= 0x0f;
        swapf   accumul,w      ;
        iorwf   char_io,w      ;
        movwf   accumul        ; accumul = (accumul << 4) | char_io;
        clrf    char_io        ; char_io = 0;
        zOS_RFI                ; break;

monchr8
        movf    char_io,w      ; } else if (char_io <= 9) { //dec only<=99?
        andlw   0xf0           ; uint16_t sum;
        btfss   STATUS,Z       ; accumul <= 1;
        bra     monchr9        ; accumul |= (accumul & 0x80) ? 1 : 0;
        ; accumul <= 1;
        lslf    accumul,f      ; w = accumul; //w keeps original accumul<1
        rlf     accumul,f      ; accumul <= 1;
        movf    accumul,w      ; accumul |= (accumul & 0x80) ? 1 : 0;
        ; accumul <= 1;
        lslf    accumul,f      ; accumul |= (accumul & 0x80) ? 1 : 0;
        rlf     accumul,f      ; accumul <= 1; // accumul:accumul <= 3;
        ; if (numbase & 2) { // base 10 presumed
        lslf    accumul,f      ; sum = (accumuh<<8)+accumul + w;
        rlf     accumul,f      ; accumul = sum & 0x00ff;
        btfss   numbase,1      ; accumul = sum >> 8;
        bra     $+4            ;
        addwf   accumul,f      ; sum = (accumuh<<8)+accumul + char_io&0x0f;
        movlw   0              ; accumul = sum & 0x00ff;
        addwfc  accumul,f      ; accumul = sum >> 8;
        movf    char_io,w      ; break;
        andlw   0x0f           ; }
        addwf   accumul,f      ; } // if ()
        movlw   0              ; char_io = 0;
        addwfc  accumul,f      ; zOS_AR1 = accumul;
        clrf    char_io        ; if (isr) goto isr; // with zOS_AR1=accumul
        zOS_RFI

```

monchr9

```

        movf    accumul,w      ; } // switch ()
        movwf   zOS_AR1        ; } // if ()
        pagesel isr
        if (isr)
            goto isr           ; char_io = 0; // unhandled
        else
            clrf    char_io      ; zOS_RFI(); // reached only if isr == 0
            zOS_RFI
        endif

        ;;
        monprmp
        movf    1+destreg,w    ;monprmp:
        movwf   accumul        ; accumul = destreg>>8;
        iorwf   destreg,w      ; if (destreg) { // prompt with destreg if nonzero
        pagesel monhex
        btfsc   STATUS,Z       ; monhex(zos_job, p0);
        bra     $+6            ; accumul = destreg & 0xff;
        call    monhex         ; monlsb(zos_job, p0);
        movf    destreg,w      ; }
        movwf   accumul        ;monlast: zOS_ACC(&accumul,&numbase); zOS_RFI();
        pagesel monlsb
        call    monlsb         ; char_io = 0;
        zOS_ACC  accumul,numbase

        monlast
        clrf    char_io        ;} // zOS_MON()
        zOS_RFI

        endmon
        endm

        zOS_CLC macro p,ra,rt,h,pi,isr;inline void zOS_CLC(int8_t p, int8_t ra, int8_t
        local   endclc,clcistr,clcpmp,endclc

        zOS_MON p,ra,rt,h,pi,clcistr
        pagesel endclc
        goto    endclc         ; rt, int8_t* h, int8_t pi, void(*isr)() {

        local   p0,p1,wrap,t0scale,isradrl,isradrh,tskadrl,tskadrh,optadrl
        local   optadrh,accumul,accumuh,numbase,destreg,destreh,char_io,buf,max

        ;; 0x20~24 reserved for zOS_CON
        p0      set    0x20
        p1      set    0x21
        wrap    set    0x22
        t0scale set    0x23

        ;; 0x24~28 reserved for zOS_INP
        isradrl set    0x24
        isradrh set    0x25
        tskadrl set    0x26
        tskadrh set    0x27

        ;; 0x28~2F reserved for zOS_MON and derivations e.g. zOS_MAN
        optadrl set    0x28
        optadrh set    0x29
        accumul set    0x2a
        accumuh set    0x2b
        numbase set    0x2c
        destreg set    0x2d
        destreh set    0x2e
        char_io set    0x2f
        buf     set    0x30
        max     set    0x70

        ;copy the preceding lines rather than including this file, as definitions for
        ;zOS_MON()-derived macros referring to these local variables wouldn't open it
        ;until expansion and would throw an undefined-var error during the processing

        clcistr

```

```

        movf    zOS_AR0,w    ; switch (char_io = zOS_AR0) {
        movwf   char_io      ;
        xorlw   '+'          ;
        btfss   STATUS,Z     ;
        bra     clcchr2      ; case '+' : // 16-bit signed/unsigned add

        movf    accumul,w    ;
        addwf   destreg,f    ;
        movf    accumuh,w    ;
        addwfc  l+destreg,f   ; destreg += (accumuh << 8) | accumul;
        bra     clcprmp      ; break;

clcchr2
        movf    char_io,w    ;
        xorlw   '-'          ;
        btfss   STATUS,Z     ;
        bra     clcchr3      ; case '-' : // 16-bit signed/unsigned subtract

        movf    accumul,w    ;
        subwf   destreg,f    ;
        movf    accumuh,w    ;
        subwfc  l+destreg,f   ; destreg -= (accumuh << 8) | accumul;
        bra     clcprmp      ; break;

clcchr3
        movf    char_io,w    ;
        xorlw   '*'          ;
        btfss   STATUS,Z     ;
        bra     clcchr4      ; case '*' : // 8-bit by 8-bit unsigned multiply
        clrf    zOS_AR0      ; // invoker of macro must implement zos_mac()
        clrf    zOS_AR1      ; // input arg zOS_AR1:zOS_AR0 (accumulator)
        movf    accumul,w    ; // zOS_AR2 (factor 1)
        movwf   zOS_AR2      ; // zOS_AR3 (factor 2)
        movf    destreg,w    ; // output arg zOS_AR1:zOS_AR0 (product)
        movwf   zOS_AR3      ; zOS_AR0 = (uint16_t) 0;
        movwf   zOS_AR2      ; zOS_AR2 = accumul & 0x00ff;

        zOS_LOC FSR0,zOS_JOB,char_io
        pagesel zos_mac
        call    zos_mac      ; zOS_AR3 = destreg & 0x00ff;
        movf    zOS_AR0,w    ; fsr0 = &char_io; // temp register (as INDF0)
        movwf   destreg      ; zos_mac(&zOS_AR0 /* += */,
        movf    zOS_AR1,w    ; &zOS_AR2 /* * */ , &zOS_AR3, fsr0);
        movwf   l+destreg    ; destreg = (uint16_t) zOS_AR0;
        bra     clcprmp      ; break;

clcchr4
        movf    char_io,w    ;
        xorlw   '/'          ;
        btfss   STATUS,Z     ;
        bra     clcchr5      ; case '/' : // 15-bit by 8-bit unsigned divide
        movf    destreg,w    ; // invoker of macro must implement zos_div()
        movwf   zOS_AR0      ; // input arg zOS_AR1:zOS_AR0 (dividend)
        movf    l+destreg,w  ; // zOS_AR2 (divisor)
        andlw   0x7f        ; // output arg zOS_AR1:zOS_AR0 (quotient/exc)
        movwf   zOS_AR1      ; zOS_AR0 = (uint16_t) destreg & 0x7fff;
        movf    accumul,w    ; zOS_AR2 = accumul & 0xff;
        movwf   zOS_AR2      ; fsr0 = &char_io; // temp register (as INDF0)

        zOS_LOC FSR0,zOS_JOB,char_io
        pagesel zos_div
        call    zos_div      ; zos_div(&zOS_AR0 /* /= */,
        movf    zOS_AR0,w    ; &zOS_AR2, &zOS_AR3/*scratch*/, fsr0);
        movwf   destreg      ;
        movf    zOS_AR1,w    ;
        movwf   l+destreg    ; destreg = (uint16_t) zOS_AR0;
        bra     clcprmp      ; break;

clcchr5
        movf    char_io,w    ;
        xorlw   '^'          ;

```

```

    call    monhex          ; accumuh = destreg & 0xff; monlsb(zos_job, p0);
    movf    destreg,w       ; moncrlf(zos_job, p0);
    movwf   accumuh        ; clclast:
    pagesel monlsb
    call    monlsb          ; zOS_ACC(&accumul,&numbase); zOS_RFI();
    pagesel moncrlf
    call    moncrlf         ; char_io = 0;
    zOS_ACC accumul,numbase
clclast
    clrf    char_io        ;}
    zOS_RFI
endclc
    endm

```

```

zos_MAN macro p, rat, rts, hb, pin; inline void zOS_MAN(int8_t p, int8_t rat,
    local mantask, manisr, manchr, manchr0, reenabl, manchr1, manchr2, manchr3
    local manchr4, manchr5, manchr6, manchr7, manchr8, manchr9, mannone, jobinfo
    local crlf, stkinfo, stkloop, endman

    local p0, p1, wrap, t0scale, isradrl, isradrh, tskadrl, tskadrh, optadrl
    local optadrh, accumul, accumuh, numbase, destreg, destreh, char_io, buf, max

```

```

    zOS_MON p, rat, rts, hb, pin, 0
    movlw   low mantask      ; int8_t* hb, int8_t pin) {
    movwi   FSR1++          ; zOS_MON(p, ra, rt, h, pi, manisr); //fsr0=swi, l=adr
    movlw   high mantask     ; optadrl = mantask & 0x00ff;
    movwi   FSR1++          ; optadrh = mantask >> 8;
    pagesel endman
    goto    endman          ;}

```

```

    ; 0x20~24 reserved for zOS_CON

```

```

p0      set    0x20
p1      set    0x21
wrap    set    0x22
t0scale set    0x23

```

```

    ; 0x24~28 reserved for zOS_INP

```

```

isradrl set    0x24
isradrh set    0x25
tskadrl set    0x26
tskadrh set    0x27

```

```

    ; 0x28~2F reserved for zOS_MON and derivations e.g. zOS_MAN

```

```

optadrl set    0x28
optadrh set    0x29
accumul set    0x2a
accumuh set    0x2b
numbase set    0x2c
destreg set    0x2d
destreh set    0x2e
char_io set    0x2f
buf      set    0x30
max      set    0x70

```

```

;copy the preceding lines rather than including this file, as definitions for
;zOS_MON()-derived macros referring to these local variables wouldn't open it
;until expansion and would throw an undefined-var error during the processing

```

```

mantask
    movf    zOS_JOB,w       ;int8_t mantask(void) { //destreg, accumul, char_io
    movwf   BSR             ; bsr = zos_job; // to access char_io
    movf    char_io,w       ; if (char_io == 0)
    btfsc   STATUS,Z        ; return 0; // back to zOS_CON task
    return  ; switch (char_io) {

    xorlw   'G'             ;
    btfss   STATUS,Z        ; caseG:

```

```

    bra     manchr          ; case 'G': // Generate a fork/duplicate of job
    clrf    char_io        ; char_io = 0; // presume failure, so no retry

    movf    accumul,w       ; if (accumul == 0)
    btfsc   STATUS,Z        ; return 0;
    return  ; zOS_ARG(0, accumul);
    zOS_ARG 0
    zOS_ACC accumul,numbase
    movlw   'J'             ; zOS_ACC(&accumul, &numbase); // reset
    movwf   char_io        ; if (zOS_SWI(zOS_FRK))
    zOS_SWI zOS_FRK
    andlw   0x00            ; goto caseJ; // success, prints in job list
    btfsc   STATUS,Z        ; else
    clrf    char_io        ; break; // failure, drop to end of switch()

```

```

manchr
    movf    char_io,w       ;
    xorlw   'H'             ;
    btfss   STATUS,Z        ; caseH:
    bra     manchr0        ; case 'H': // find jobs by Handle (start addr)
    clrf    char_io        ; char_io = 0;

```

```

    movf    accumul,w       ; if (accumul == 0)
    iorwf   accumuh,w       ;
    btfsc   STATUS,Z        ; return 0;
    return  ; zOS_ARG(0, accumul);
    movf    accumul,w       ;
    zOS_ARG 0
    movf    accumuh,w       ;
    zOS_ARG 1
    zOS_ACC accumul,numbase
    movlw   'J'             ; zOS_ACC(&accumul, &numbase);
    movwf   char_io        ; if (zOS_SWI(zOS_FND))
    zOS_SWI zOS_FND
    andlw   0x00            ; goto caseJ; // FIXME: table, from match down
    btfsc   STATUS,Z        ; else
    clrf    char_io        ; break;

```

```

manchr0
    movf    char_io,w       ;
    xorlw   'I'             ;
    btfss   STATUS,Z        ; caseI:
    bra     manchr1        ; case 'I': // send a software Interrupt > 7
    clrf    char_io        ; char_io = 0; // with destreg zOS_AR1:zOS_AR0

    movf    destreg,w       ; zOS_ARG(0, destreg);
    zOS_ARG 0
    movf    1+destreg,w     ; zOS_ARG(1, destreh);
    zOS_ARG 1
    movlw   0xf8            ; zOS_ACC(&accumul, &numbase); // reset
    andwf   accumul,w       ;
    zOS_ACC accumul,numbase
    btfsc   STATUS,Z        ; if (accumul) {
    bra     reenabl        ; int w = zOS_SWI(accumul); // disable again
    movlp   0               ; INTCON &= ~(1<<GIE); // for zOS_AR and _BUF()
    call    0x02            ; zOS_ARG(1, w);
    bcf     INTCON,GIE      ; zOS_ARG(0, 0);
    clrf    zOS_AR1         ; zOS_BUF(zos_job, p0); // print hex SWI result
    xorwf   zOS_AR1,f       ; zOS_ENA();
    xorwf   zOS_AR0,f       ; goto caseJ;
    zOS_BUF zOS_JOB,buf,p0
    movlw   'J'             ; } else
    movwf   char_io        ; zOS_ENA(); break;
    reenabl
    zOS_ENA

```

```

manchr1
    movf    char_io,w       ;
    xorlw   'J'             ;

```

```

        btfss    STATUS,Z      ; caseJ:
        bra     manchr2        ; case 'J': // List struct for all running jobs

        decf     accumul,w      ; // keep char_io='S' until last job line prints
        andlw    0x07           ;
        btfsc    WREG,2         ; if ((accumul < 1) || (accumul > 5))
        movlw    zOS_NUM-1     ;
        addlw    0x01           ;
        movwf    accumul        ; accumul = zOS_NUM;
        bcf      INTCON,GIE     ; INTCON &= ~(1<<GIE); // to keep p0==p1 atomic
        pagesel  jobinfo        ;
        movf     p0,w           ;
        xorwf    p1,w           ; if (p0 == p1)
        btfsc    STATUS,Z      ; return jobinfo(); // will decrement accumul
        goto     jobinfo        ; zOS_ENA(); // re-enable interrupts if p0!=p1
        zOS_ENA
        retlw    0              ; return 0; // try again after caller advances p0

manchr2
        movf     char_io,w      ;
        xorlw    'K'           ;
        btfss    STATUS,Z      ; caseK:
        bra     manchr3        ; case 'K': // Kill a single job (# mandatory)
        clrf     char_io        ; char_io = 0;

        movf     accumul,w      ; if (accumul == 0)
        btfsc    STATUS,Z      ; return 0;
        return   ; zOS_ARG(0, accumul);
        zOS_ARG 0
        zOS_ACC  accumul,numbase
        movlw    'J'           ; zOS_ACC(&accumul, &numbase);
        movwf    char_io        ; zOS_SWI(zOS_END); // listed indicates failure
        zOS_SWI  zOS_END

;;; FIXME: put J at bottom so K onward don't pay a performance penalty awaiting

manchr3
        movf     char_io,w      ;
        xorlw    'L'           ;
        btfss    STATUS,Z      ; caseL:
        bra     manchr4        ; case 'L': // Launch a fresh instance of a job
        clrf     char_io        ; char_io = 0;

        movf     accumul,w      ; if (accumul == 0)
        btfsc    STATUS,Z      ; return 0;
        return   ; zOS_ARG(0, accumul);
        zOS_ARG 0
        zOS_ACC  accumul,numbase
        movlw    'J'           ; zOS_ACC(&accumul, &numbase); // reset
        movwf    char_io        ; if ((w = zOS_SWI(zOS_FRK)) != 0) {
        zOS_SWI  zOS_FRK        ;
        andlw    0x00           ; zOS_ARG(0,w); zOS_SWI(zOS_RST);
        btfsc    STATUS,Z      ; goto caseJ; // success, prints in job list
        clrf     char_io        ; } else
        zOS_ARG 0
        zOS_SWI  zOS_RST        ; break; // failure, drop to end of switch()

manchr4
        movf     char_io,w      ;
        xorlw    'N'           ;
        btfss    STATUS,Z      ; caseN:
        bra     manchr5        ; case 'N': // New (parameterless) job at addr

        movf     accumul,w      ;
        movwf    FSR0L          ;
        movf     accumul,w      ;
        movwf    FSR0L          ;
        clrw     ;
        zOS_ARG 0
        zOS_ARG 1

```

```

        zOS_ARG 2
        zOS_ARG 3
        zOS_SWI  zOS_NEW
        zOS_ARG 0
        zOS_BUF  zOS_JOB,buf,p0
        movlw    'J'           ;
        movwf    char_io        ;

        movf     accumul,w      ; if (accumul == 0)
        btfsc    STATUS,Z      ; return 0;
        return   ; zOS_ARG(0, accumul);
        zOS_ARG 0
        zOS_ACC  accumul,numbase
        movlw    'J'           ; zOS_ACC(&accumul, &numbase);
        movwf    char_io        ; if ((w = zOS_SWI(zOS_SLP)) != 0) {
        zOS_SWI  zOS_SLP        ;
        andlw    0xff           ; accumul = w;
        movwf    accumul        ; goto caseJ;
        btfsc    STATUS,Z      ; } else
        clrf     char_io        ; break;

manchr5
        movf     char_io,w      ;
        xorlw    'P'           ;
        btfss    STATUS,Z      ; caseP:
        bra     manchr6        ; case 'P': // Pause job by putting it to Sleep
        clrf     char_io        ; char_io = 0;

        movf     accumul,w      ; if (accumul == 0)
        btfsc    STATUS,Z      ; return 0;
        return   ; fsrl = 0x10 * (1 + accumul) + zOS_PCH;
        movlw    'J'           ;
        movwf    char_io        ;
        zOS_MEM  FSR1,accumul,zOS_PCH
        movf     INDF1,w        ; if (*fsrl) { // is a valid (PCH not 0x00) job
        btfsc    STATUS,Z      ; *fsr |= 0x80;
        clrf     char_io        ; goto caseJ;
        iorlw    0x80           ; } else {
        movf     INDF1,f        ;
        btfss    STATUS,Z      ;
        movwf    INDF1          ; zOS_ACC(&accumul, &numbase);
        btfsc    STATUS,Z      ; break; // only clear accumul if not caseJ
        bra     manchr6        ; }
        zOS_ACC  accumul,numbase

manchr6
        movf     char_io,w      ;
        xorlw    'Q'           ;
        btfss    STATUS,Z      ; caseQ:
        bra     manchr7        ; case 'Q': // Quit without wake (off)
        clrf     char_io        ; char_io = 0;

        bcf      WDTCN,SWDTEN   ; WDTCN &= ~(1<<SWDTEN);
        movf     accumul,f      ;
        btfss    STATUS,Z      ; if (accumul)
        sleep    ; sleep(); // never wakes up

manchr7
        movf     char_io,w      ;
        xorlw    'R'           ;
        btfss    STATUS,Z      ; caseR:
        bra     manchr8        ; case 'R': // Resume a pause/asleep job
        clrf     char_io        ; char_io = 0;

        movf     accumul,w      ; if (accumul == 0)
        btfsc    STATUS,Z      ; return 0;
        return   ; fsrl = 0x10 * (1 + accumul) + zOS_PCH;
        movlw    'J'           ;
        movwf    char_io        ; if (*fsrl &= ~(1<<zOS_WAI)) {

```

```

zos_MEM FSR1,accumul,zOS_PCH
movlw 0x7f ; goto caseJ; // valid job won't be 0 or 0x80
andwf INDF1,f ; } else {
btfss STATUS,Z ; zOS_ACC(&accumul, &numbase);
bra manchr8 ;
zos_ACC accumul,numbase
clrf char_io ; break; // only clear accumul if not caseJ

manchr8
movf char_io,w ; }
xorlw 'S' ;
btfss STATUS,Z ;
bra manchr9 ; case 'S': // Stack dump is actually scratch
clrf char_io ; char_io = 0; // always succeeds, no arg

decf accumul,w ; // keep char_io='J' until last job line prints
andlw 0x07 ;
btfsc WREG,2 ; if ((accumul < 1) || (accumul > 5))
movlw zOS_NUM-1 ;
addlw 0x01 ;
movwf accumul ; accumul = zOS_NUM;
bcf INTCON,GIE ; INTCON &= ~(1<<GIE); // to keep p0==p1 atomic
pagesel stkinf
movf p0,w ;
xorwf pl,w ; if (p0 == p1)
btfsc STATUS,Z ; return jobinfo(); // will decrement accumul
goto stkinf ; zOS_ENA(); // re-enable interrupts if p0!=p1
zos_ENA
retlw 0 ; return 0; // try again after caller advances p0

manchr9
movf char_io,w ;
xorlw 'Z' ;
btfss STATUS,Z ;
bra mannone ; case 'Z': // go to low-power Zz mode for time
clrf char_io ; char_io = 0;

bsf WDTCON,SWDTEN ; if (w = accumul<1) { // WDT prescaler
lslf accumul,w ; w |= 1<<SWDTEN; // enable the wakeup
btfsc STATUS,Z ;
bra mannone ;
iorlw 1<<SWDTEN ;
movwf WDTCON ;
sleep ; break; // wakes up according to prescaler

mannone
retlw 0 ; } return 0; // naught to do }

;guaranteed to arrive with p0=p1, interrupts off and in the correct bank
stkinf
movf wrap,f ;int8_t stkinf(void) {
movwf p0 ; p0 = p1 = wrap;
movwf pl ;
movlw low zOS_STK ;
movwf FSR0L ;
movlw high zOS_STK ;
movwf FSR0H ;
decf accumul,w ;
brw ;
addfsr FSR0,6 ;
addfsr FSR0,6 ;
addfsr FSR0,6 ;
addfsr FSR0,6 ; fsr0 = zOS_STK + 6 * (5 - accumul);
zos_LOC FSR1,zOS_JOB,buf
movlw '\r' ; fsr1 = (zos_JOB << 7) + buf;
movwi FSR1++ ;
movlw '\n' ;
movwi FSR1++ ;
movlw '-' ;

movwi FSR1++ ;
movlw '-' ;

stkloop
movlw ' ' ;
movwi FSR1++ ; pl += sprintf(pl, " %04X", *((int*) fsr0));
moviw --FSR0 ;
movwi FSR1++ ;
moviw --FSR0 ;
movwi FSR1++ ;
decfsz accumuh,f ;
bra stkloop ; }

movf FSR1L,w ;
movwf pl ; w = accumul--; // return with w as nonzero job
movf accumul,w ; if (accumul == 0)
decf accumul,f ; char_io = 0; // final row in table was printed
btfsc STATUS,Z ; zOS_ENA(); // interrupts back ON!
clrf char_io ; return w;
zos_ENA
return ;}

;guaranteed to arrive with p0=p1, interrupts off and in the correct bank
jobinfo
movf wrap,f ;int8_t jobinfo(void) {
movwf p0 ; p0 = p1 = wrap;
movwf pl ; fsr0 = 0x10 * (1 + accumul); //FIXME: 2+
zos_MEM FSR0,accumul,0
zos_LOC FSR1,zOS_JOB,buf
movlw '\r' ; fsr1 = (zos_JOB << 7) + buf;
movwi FSR1++ ;
movlw '\n' ;
movwi FSR1++ ;
movf accumul,w ; // print this job number 5/4/3/2/1
zos_HEX
movwi FSR1++ ; pl += sprintf(pl, "\r\n%1X", accumul);

moviw zOS_HDH[FSR0] ;
andlw 1<<zOS_PRB ;
movlw ':' ; // print '*' if the job is privileged else ':'
btfsc STATUS,Z ;
movlw '*' ; pl += sprintf(pl, "%c", (zos_HDH[fsr0] &
moviw FSR1++ ; (1<<zOS_PRB)) ? '*' : ':');

zos_IHF zOS_HDH,FSR0,FSR1
zos_IHF zOS_HDL,FSR0,FSR1
movlw ' ' ;
movwi FSR1++ ;
movlw 'P' ; // print the 4-hex-digit header then PC
movwi FSR1++ ;
movlw 'C' ; pl += sprintf(pl, "%04X PC",
movwi FSR1++ ; (zos_HDH[fsr0] << 8) + zOS_HDL[fsr0]);

moviw zOS_PCH[FSR0] ;
andlw 1<<zOS_WAI ;
movlw '=' ; // print '=' if the job is sleeping else 'z'
btfsc STATUS,Z ;
movlw 'z' ; pl += sprintf(pl, "%c", (zos_PCH[fsr0] &
movwi FSR1++ ; (1<<zOS_WAI)) ? 'z' : ':');

zos_IHF zOS_PCH,FSR0,FSR1
moviw zOS_PCH[FSR0] ; // drop out after PCH if 0 (job is deleted)
btfsc STATUS,Z ; pl += sprintf(pl, "%02X", zOS_PCH[fsr0]);
bra crlf ; if (zos_PCH[fsr0] & 0xff00) {
zos_IHF zOS_PCL,FSR0,FSR1

```



```

        movlw    ' '          ; // print the low byte of program counter
        movwi    FSR1++        ; pl += sprintf(pl, "%02X", zOS_PCL[fsr0]);
        moviw    zOS_ISH[FSR0] ;
        btfss    STATUS,Z      ; // drop out after PCL if no interrupt routine
        bra      crlf          ; if (zOS_ISH[fsr0] & 0xff00) {
        movlw    'I'          ;
        movwi    FSR1++        ;
        movlw    'S'          ;
        movwi    FSR1++        ;
        movlw    'R'          ;
        movwi    FSR1++        ;
        movlw    '@'          ;
        movwi    FSR1++        ; // print ISR@ then 4-hex-digit routine addr
        zOS_IHF zOS_ISH,FSR0,FSR1
        zOS_IHF zOS_ISR,FSR0,FSR1
        movlw    '('          ; pl += sprintf(pl, " ISR@%04X",
        movwi    FSR1++        ;         (zOS_ISH[fsr0] << 8) + zOS_ISR[fsr0]);
        movlw    'h'          ;
        movwi    FSR1++        ;
        movlw    'w'          ;
        movwi    FSR1++        ;
        zOS_IHF zOS_HIM,FSR0,FSR1
        movlw    's'          ;
        movwi    FSR1++        ;
        movlw    'w'          ;
        movwi    FSR1++        ; // print (hw HwIMask sw SwIMask) scrunched up
        zOS_IHF zOS_SIM,FSR0,FSR1
        movlw    ')'          ; pl += sprintf(pl, "(hw%02Xsw%02X)",
        movwi    FSR1++        ;         zOS_HIM[fsr0], zOS_SIM[fsr0]);
crlf
        movlw    '\r'          ; }
        movwi    FSR1++        ; }
        movlw    '\n'          ; // print a second \r\n, double-spacing table
        movwi    FSR1++        ; pl += sprintf(pl, "\r\n");

        movf     FSR1L,w        ;
        movwf    pl             ; w = accumul--; // return with w as nonzero job
        movf     accumul,w      ; if (accumul == 0)
        decf     accumul,f      ; char_io = 0; // final row in table was printed
        btfsc    STATUS,Z      ; zOS_ENA(); // interrupts back ON!
        clrf     char_io        ; return w;
        zOS_ENA
        return                ;}
endman
endm

```