```
;;
;; iopuart.s
;;
;; Copyright (c) 1998 Patriot Scientific Corporation.  All rights reserved.
;;

;;---------------------------------------------------------------------------
;;
;; I/O processor UART
;; Baud rate selectable in software
;;
;; Starts when IOP is reset (see init_uart())
;; Performs necessary DRAM refresh cycles
;; Baud rate can be easily changed (see init_uart)
;;
;; Uses -IN1 (input 1) as serial data input.
;; Uses OUT1 (output 1) as serial data output.
;; Uses interrupt 4 for transmit bit interrupt.
;; Uses interrupt 1 for receive bit interrupt.
;; Uses registers g7, g13, g14, g15.
;;
;;---------------------------------------------------------------------------

        .opt    noincl
        .include "onchip.def"
        .include "sysmem.def"

        .text   iop_uart, iop

        ; Timing on the IOP is performed by counting memory-access and program-
        ; -execution cycles.  If any memory-access timings are changed, or if
        ; the CPU oscillator frequency is changed, the values below will need
        ; to be adjusted. The "+ 2" accounts for the slot-check time.
        ; Group 0 is the memory containing the IOP program.
        ; Group 1 is the group which is refreshed (i.e., controls refresh timing).

        CAS0 = 4 + 2                    ; cycles per CAS cycle for group 0
        RAS0 = 5 + CAS0                 ; cycles per RAS cycle for group 0
        CAS1 = 5 + 2                    ; cycles per CAS cycle for group 1
        RAS1 = 6 + CAS1                 ; cycles per RAS cycle for group 1

        proc_freq = 40000000 * 2        ; 40 MHz external clock; proc_freq
                                        ; (X2 clock) is twice the external clock

        ; Serial data is sent with one start bit, N data bits, and
        ; one or more stop bits.  If desired, parity must be treated as an
        ; extra data bit and handled in software.

        ; The IOP measures time intervals in whole numbers of "slices".
        ; For accurate serial timing, the slice rate should be an integer
        ; multiple of the bit rate.  It should be at least eight times
        ; the fastest bit rate used, so that the worst case timing error
        ; is only 1/8 of a bit.  Note that setting max_baud_rate too high
        ; will result in a slice_delay too small for useful work, or
        ; worse yet, a negative (impossible) slice_delay.

        max_baud_rate = 9600            ; baud rate (bits per second)
        slice_rate = max_baud_rate * 8  ; IOP wakes up 8 times per bit

        cycles_per_slice = proc_freq / slice_rate  ; number of X2 cycles
                                        ; in one slice period (1/8th of a bit)

        ; A typical DRAM requires 512 refreshes every 8 ms period.
        ; To ensure adequate DRAM refresh, a "slice" must occur at least
        ; once every 8 msec.  This corresponds to a bit period of 64 msec,
        ; so max_baud_rate must be set to AT LEAST 16 baud.

        DRAM_refreshes = 512            ; this many refreshes,
        DRAM_msec      = 8              ; per this many msec

        ; Three refreshes will be performed in each iteration of
        ; the micro loop.
        refreshes_per_iteration = 3

        ; Calculate the number of iterations of the micro loop as follows:
        ;
        ;           #refreshes    #msec
        ;           ----------  x -----
        ;             period       sec              #iterations
        ;    ---------------------------------  =  -----------
        ;       #msec    #slices   #refreshes          slice
        ;       ------ x ------- x ----------
        ;       period     sec      iteration
        ;
        ; Because 'iterations' has a fractional part which is truncated
        ; by the divide, add 1 to round up to the next whole number.

        iterations = 1 + ((DRAM_refreshes * 1000) / (DRAM_msec * slice_rate * refreshes_per_iteration))

        ; The total time for each slice is the sum of the delay (g7),
        ; the refresh microloop, and the other processing.
        ; Thus if we know the number of X2 cycles for one slice period,
        ; and we know how many "overhead" cycles are required in each
        ; slice, we can compute the required number of delay cycles.

        refresh_overhead = (RAS1 + RAS1 + RAS1 + 1) * iterations
        other_overhead = (6*CAS0) + 6 + RAS0
        total_overhead = refresh_overhead + other_overhead

        slice_delay = cycles_per_slice - total_overhead

;;------------------------------------------------------------------------------
;;
;; IOP_uart -- IOP program to perform UART timing
;;
;; The IOP UART divides each serial bit period into at least 8 slices.
;; The IOP "wakes up" for each slice to possibly interrupt the MPU
;; for a receive bit, and possibly interrupt the MPU for a transmit bit.
;; Thus a timing resolution of 1/8 of a bit (or better) is achieved.
;;
;; When the MPU detects the leading edge of a start bit, it sets the
;; Rx_slice_counter (register g13) to one half bit period.  It then
;; checks that the start bit is valid.  If so, the MPU sets the
;; Rx_slice_counter to one bit period; when the register is decremented
;; to zero, the MPU is interrupted to sample the serial data (in the
;; center of the bit).
;;
;; When the MPU wants to output a serial byte, it outputs the
;; start bit and then sets the Tx_slice_counter (register g14)
;; to one bit period.  After this register is decremented to zero,
;; the MPU is interrupted to output another serial bit.
;;
;; During each slice delay, the MPU must ensure that the DRAM
;; is adequately refreshed.  This is accomplished by doing a
;; burst of refreshes during each slice period.
;;
;; g7 = delay count for slice period
;; g15 = refresh loop counter
;; g14 = tx slice counter - loaded by MPU
;; g13 = rx slice counter - loaded by MPU
;;
;;------------------------------------------------------------------------------

IOP_uart::              ; Cycle counts assume this code executes from
                        ; group 0, and group 1 controls refresh timing
        ld  #slice_delay,g7
        ld  #-1,g14             ; Set Tx slice counter to a safe (large) value
        ld  #-1,g13             ; Set Rx slice counter to a safe (large) value

        ; Wait for first start bit sample (one slice period)

start0::                ; CAS0 (instruction fetch)
        delay   g7      ; 2 + g7
        ld  #iterations,g15  ; RAS0
        nop             ; 1
        nop             ; 1
```

```
                        ; CAS0 (instruction fetch)
        refresh         ; RAS1  \
        refresh         ; RAS1   \ * g15 iterations
        refresh         ; RAS1   /
        mloop   g15     ; 1     /

        ; Check if time to generate Tx interrupt

                        ; CAS0 (instruction fetch)
                        ; -noskip-  -skip-
        dskipz  g14     ; 1         fetch at target    (check Tx counter)
        jump.3  s0_notx ; fetch at target

                        ;           CAS0 (instruction fetch)
        int     4       ;           1
        jump.3  s0_tx   ;           fetch at target

s0_notx::               ; CAS0 (instruction fetch)
        jump    s0_tx   ; fetch at target

s0_tx::

        ; Check if time to generate Rx interrupt

                        ; CAS0 (instruction fetch)
                        ; -noskip-  -skip-
        dskipz  g13     ; 1         fetch at target    (check Rx counter)
        jump.3  s0_norx ; fetch at target

                        ;           CAS0 (instruction fetch)
        int     1       ;           1
        jump.3  start0  ;           fetch at target

s0_norx::               ; CAS0 (instruction fetch)
        jump.3  start0  ; fetch at target

s0_rx::


;;---------------------------------------------------------------------------
;;
;; tx_isr -- transmit bit interrupt service routine  ("single buffered")
;;
;; This interrupt occurs when one bit period haa elapsed since the
;; last transmit interrupt, i.e., when it is time to send another bit.
;; The service routine must output the next bit (from utx_data)
;; and then reset the transmit slice counter (g14) to N
;; (according to the desired baud rate).
;;
;; When utx_bit_count is nonzero, bits remain to be sent
;; from utx_data.  Send the LSB of utx_data, shift utx_data right,
;; and decrement utx_bit_count.
;;
;; When the (last) stop bit is sent, utx_bit_count will be decremented
;; from 1 to 0.  We must delay one more bit period before
;; we can accept new transmit data, to allow the stop bit to complete.
;;
;; When the interrupt occurs and utx_bit_count is 0, the stop bit is
;; complete.  Leave the output unchanged, leave utx_bit_count unchanged,
;; and clear utx_data.  The high-level routines will recognize this as
;; 'transmitter empty' and will load a new character.
;;
;; After all data bits have been output reset the transmit slice counter
;; to -1.  This will ensure that an interrupt does not occur for 2**32-1
;; slices  (55924 seconds, or about 15.5 hours, at max_baud_rate=9600).
;; Should that time elapse, the interrupt service routine will see
;; utx_bit_count = 0, and will simply reset the slice counter.
;;
;; This routine is installed in the interrupt-vector table by init_uart().
;;
;; Notes:
;;
;; It is necessary to save/restore mode because the GRS bits are
;; affected by 'shr'.
;;
;;-----------------------------------------------------------------------------

        .text

        .lcomm  utx_data,4              ; data byte being transmitted
        .lcomm  utx_bit_count,4         ; number of bits left to transmit

        ; Bit counter is zero.  Leave output unchanged.
        ; Reset slice counter to -1, and utx_data to 0.
        ; ( mode adr count ) is on the operand stack.

no_tx_data::
        push.n  #0                      ; clear utx_data to signal
        push    #utx_data               ;  transmitter empty
        st      []
        pop

        push.n  #-1                     ; very large slice count
        pop     g14                     ;  to forestall tx interrupts
        pop     mode
        reti

        ; Interrupt entry point
tx_isr::
        push    mode
        push    #utx_bit_count          ; number of bits still to be sent
        ld      []
        bz      no_tx_data              ; branch backwards so a 3-bit
                                        ; offset can be used

        ; At least one bit remains to be sent.
        ; ( mode ) is on the operand stack.

        push    #slices_per_bit         ; Reset slice counter to one bit.
        ld      []
        pop     g14

        push    #utx_data               ; Get current data cell.
        push
        ld      []                      ;       ( -- mode adr data )

        push                            ; Output LSB to OUT1.
        push.n  #1
        and
        push    #io1out_i
        sto.i   []
        pop

        shr     #1                      ; Shift data 1 position right,
        xcg
        st      []                      ;   and store back in utx_data.
        pop

        push    #utx_bit_count          ; number of bits still to be sent
        push

        ld      []                      ; Decrement bit counter.
        dec     #1
        xcg
        st      []
        pop

        pop     mode
        reti


;;-----------------------------------------------------------------------------
;;
;; char uart_tx(char c) -- send a character
;;
;; To send a data byte, put the pattern  ..11nnnnnnnn0 into utx_data,
;; and the value total_bits+1 into utx_bit_count, then set the transmit
;; slice counter (g14) to 1.  This will ensure that an interrupt is
;; generated at the next slice interval, and the tx_isr routine will
;; output the start bit ('0').  Subsequent bits will output the data
;; lsb first, then the desired number of stop bits.
;;
;; Note that this routine does NOT first test that the transmit buffer
;; is empty.  The calling program must do this, with uart_txready().
;;
;;------------------------------------------------------------------------------

        .export _uart_tx

_uart_tx:: push                 ; Character 'c' is on the operand stack.
                                ; Push an extra copy for return value.

        push.n  #-1             ; generate mask of 1s for stop bits
        push    #N_data_bits
        ld      []
        shift

        or                      ; OR stop bits with data bits
        shl     #1              ; shift left 1 to create '0' start bit

        push    #utx_data       ; store pattern in transmit data buffer
        st      []
        pop

        push    #N_total_bits   ; set bit count to total # of bits,
        ld      []
        inc     #1              ;   plus 1 for the start bit
        push    #utx_bit_count
        st      []
        pop

        push.n  #1              ; set slice counter to wake up tx interrupt
        pop     g14
        ret


;;------------------------------------------------------------------------------
;;
;; int uart_txready() -- returns (true) nonzero if not transmitting a character
;;
;; If the transmit buffer is active, utx_data will have a nonzero
;; value.  One bit period after the last bit is output -- i.e., after
;; the END of the stop bit(s) -- utx_data is cleared to zero.
;;
;;------------------------------------------------------------------------------

        .export _uart_txready

_uart_txready::
        push    #utx_data
        ld      []
        eqz                     ; If count=0, transmitter is ready.
        ret

;;------------------------------------------------------------------------------
;;
;; rx_start_isr  -- receive interrupt service routines
;; rx_check_isr
;; rx_bit_isr
;;
;; The receiver is initialized by enabling -IN1 as an interrupt input.
;; A 1-to-0 transition on this input (i.e., the beginning of a start bit)
;; will interrupt the MPU.  The MPU must then disable the -IN1 interrupt,
;; set the receive slice counter (r13) to half a bit period, and select the
;; rx_check_isr routine for the next interrupt.
;;
;; After half a bit period elapses, rx_check_isr checks to see if -IN1 is
;; still low.  (During the ISR, the input is not zero-persistent.)
;; If so, this is deemed a valid start bit.  urx_data is cleared,
;; urx_bit_count is set to the total number of bits to receive,
;; the slice counter is set to one full bit period, and rx_bit_isr is
;; made the interrupt routine.
;;
;; For the next N bits, rx_bit_isr occurs in the middle of the bit period.
;; The input is sampled and the bit is shifted into urx_data.
;; The rx slice counter is reset to one full bit period.
;; After the last bit (the stop bit) is sampled, the interrupt
;; vector is reset to rx_start_isr, and the -IN1 interrupt enabled.
;;
;; Note that a break (input low) condition will be seen as an endless
;; series of null (zero) characters.  This framing error will be seen
;; as a zero bit in the "stop bit" position of urx_data.
;;
;; If no data is received for 2**32-1 slices, interrupt 1 will be
;; asserted by the slice counter.  The rx_start_isr routine will see
;; an invalid start bit, and will go reset itself.
;;
;;----------------------------------------------------------------------------

        .lcomm  urx_data,4              ; data byte being shifted in
        .lcomm  urx_bit_count,4         ; number of bits left to receive
        .lcomm  urx_buffer,4            ; complete data byte received

rx_isr::

        ; Enter this routine when the leading edge of the start bit
        ; is detected.
        ; Mode must be saved becuase of the SHR instruction.

rx_start_isr::
        push    mode

        push.n  #-1             ; release persisting zero
        push    #io1in_i        ; (in case this is a false start bit)
        sto.i   []
        pop

        push.n  #0              ; disable -IN1 interrupt
        push    #io1ie_i
        sto.i   []

        shl     #1              ; clear carry (for SHR instruction)
        pop

        push    #((rx_check_isr - int1_trapv) >> 2)     ; change Rx ISR
        push    #int1_trapv
        st      []
        pop

        push    #slices_per_bit ; set Rx slice counter to half a bit period
        ld      []
        shr     #1
        pop     g13

        pop     mode
        reti

        ; Enter this routine halfway through the start bit, when
        ; the slice counter reaches zero.
        ; The mode register is not disturbed.

rx_check_isr::
        push.n  #-1             ; release persisting zero on -IN1
        push    #io1in_i
        sto.i   []
        pop

        push    #io1in_i        ; test the current input state
        ldo.i   []
        bz      valid_start

invalid_start::                 ; Not a valid start bit.  Await rx_start_isr.
        push    #((rx_start_isr - int1_trapv) >> 2)     ; change Rx ISR
        push    #int1_trapv
        st      []
        pop

        push.n  #-1             ; enable -IN1 interrupt
        push    #io1ie_i
        sto.i   []
        pop
                                ; The slice counter doesn't need to be reset.
        reti

valid_start::                   ; A valid start bit.
        push    #((rx_bit_isr - int1_trapv) >> 2)       ; change Rx ISR
        push    #int1_trapv
        st      []
        pop

        push    #slices_per_bit ; set Rx slice counter to a full bit period
        ld      []
        pop     g13

        push.n  #-1             ; pre-clear data register
        push    #urx_data
        st      []
        pop

        push    #N_total_bits   ; set rx bit counter
        ld      []
        push    #urx_bit_count
        st      []
        pop

        reti

        ; Enter this routine in the center of a bit cell.
        ; Mode must be saved becuase of the SHL/SHR instructions.
        ;
        ; The first entry to this routine is data bit 0.
        ; Subsequent data bits are shifted right into the MSB
        ; of urx_data.  After the (last) stop bit is shifted
        ; into urx_data, the result is shifted to the right of
        ; the cell and put in urx_buffer.  Then the interrupt service
        ; is reset to look for a new start bit.
        ;
        ; Note that, since the stop bit is included, the result stored
        ; in urx_buffer will be nonzero.  If no stop bit is received,
        ; a zero will be stored in urx_buffer.

rx_bit_isr::

        push    mode

        push.n  #-1             ; release persisting zero
        push    #io1in_i
        sto.i   []
        pop

        push    #io1in_i        ; get the current input state..
        ldo.i   []
        shl     #1              ; ..into carry
        pop

        push    #urx_data       ; shift it into MSB of urx_data
        push
        ld      []
        shr     #1
        xcg
        st      []              ;   (leaves urx_data on stack)

        push    #urx_bit_count  ; decrement bit counter
        push
        ld      []
        dec     #1
        xcg
        st      []
        bz      rx_all_bits     ; if zero, this was the last (stop) bit.

rx_more_bits::                  ; If more bits,
        pop                     ; discard urx_data from stack

        push    #slices_per_bit ; reset Rx slice counter,
        ld      []
        pop     g13

        pop     mode            ; and return.
        reti

rx_all_bits::                   ; If that was last bit,
        push    #urx_buffer     ; store data in result buffer
        st      []              ; Note that LSBs will be '1'
        pop

        push    #((rx_start_isr - int1_trapv) >> 2)     ; change Rx ISR
        push    #int1_trapv
        st      []
        pop

        push.n  #-1             ; enable -IN1 interrupt
        push    #io1ie_i
        sto.i   []
        pop
                                ; The slice counter doesn't need to be reset.
        pop     mode
        reti


;;-----------------------------------------------------------------------
;;
;; int uart_rx() -- returns a (nonzero) character if one has been received,
;;                  else returns zero.  The buffer is cleared.
;;
;;-----------------------------------------------------------------------

        .export _uart_rx

_uart_rx::
        push    #urx_buffer
        push

        .quad 4                 ; Force these instructions to be in a group
                                ;  so they are not interrupted:
        ld      []              ; get current contents of rx buffer
        push.n  #0              ;  .
        rev                     ;  .
        st      []              ; and clear buffer

        shl     #1              ; clear carry
        pop

        push    #0x80000020     ; compute shift count,
        push    #N_total_bits   ;   ((32 - (total_bits)) | 0x80000000)
        ld      []
        sub

        shift                   ; shift urx_data to rightmost position

        ret


;;-----------------------------------------------------------------------
;;
;; void init_uart(int baudrate, int databits, int stopbits)
;;                                              -- initialize the IOP UART
;;
;; This routine sets the IOP software reset vector to jump to the above
;; IOP clock routine, and resets the IOP (thus starting the new IOP clock).
;;
;;-----------------------------------------------------------------------
```

```
        .lcomm  slices_per_bit,4        ; derived from baud rate
        .lcomm  N_data_bits,4             ; number of data bits
        .lcomm  N_total_bits,4            ; data bits + stop bits

        .export _init_uart

_init_uart::    ; stack effect:  stopbits databits baudrate --

        ; Compute slices_per_bit = slice_rate / baudrate

        pop     g0
        push    #slice_rate
        push.n  #0
        divu
        pop                             ; discard remainder (truncate result)
        push    #slices_per_bit
        st      []
        pop

        ; Store number of data bits

        push    #N_data_bits
        st      []

        ; Store total number of data bits + stop bits

        add
        push    #N_total_bits
        st      []
        pop

        ; Indicate no bits in the transmit buffer.

        push.n  #0
        push    #utx_data
        st      []
        pop

        ; Set transmit interrupt vector to execute local service routine.

        push    #((tx_isr - int4_trapv) >> 2)   ; br.4 tx_isr
        push    #int4_trapv
        st      []
        pop

        ; Set receive interrupt vector to execute local service routine.

        push    #((rx_start_isr - int1_trapv) >> 2)     ; br.4 rx_isr
        push    #int1_trapv
        st      []
        pop

        ; Set IOP software reset vector to execute new IOP code.

        push    #(0x30000000 | (IOP_uart >> 2)) ; jump.4 IOP_uart
        push    #iop_resetv
        st      []

        ; Restart the IOP.

        push    #iopreset
        sto     []
        pop

        ; Enable -IN1 interrupt to detect start bit.

        push.n  #-1             ; release persisting zero, just in case
        push    #io1in_i
        sto.i   []
        pop

        push.n  #0
        push    #io1dmae_i
        sto.i   []
        pop

        push.n  #-1
        push    #io1ie_i
        sto.i   []
        pop

        ret
```