Operating Systems

Serial Device Driver

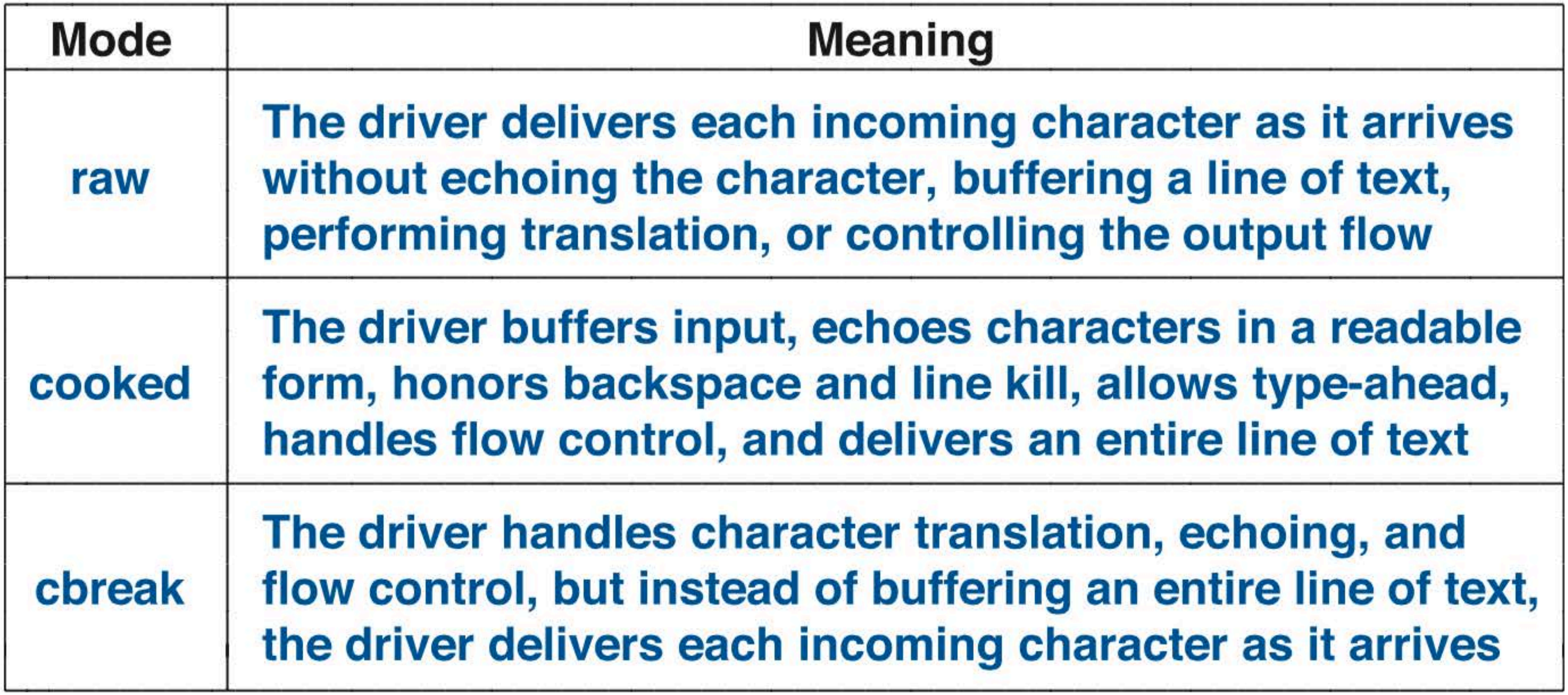
Device Drivers

- Map high level I/O functions into device-specific details
- Have an upper and lower half for interacting with the upper layers of the OS, and servicing interrupts, respectively
- This module considers the serial interface
- Recall the Universal Asynchronous Receiver/Transmitter (UART)
- Sends and receives individual bytes, one bit at a time

The Tty

- Xinu, like Unix, refers to a character-oriented text device or window as a tty
  - Derived from a Teletype device which consists of a keyboard and printer
- Serial devices have separate send and receive, but the tty notion makes them one device
- Various modes (as in Unix)
- Consider character echoing – generally on but disabled for password entry

Tty Modes



Mode	Meaning
raw	The driver delivers each incoming character as it arrives without echoing the character, buffering a line of text, performing translation, or controlling the output flow
cooked	The driver buffers input, echoes characters in a readable form, honors backspace and line kill, allows type-ahead, handles flow control, and delivers an entire line of text
cbreak	The driver handles character translation, echoing, and flow control, but instead of buffering an entire line of text, the driver delivers each incoming character as it arrives

Tty Modes

- Raw mode for file transfer, and text “GUI”
- Cooked mode enables flow control, pausing the output with control-s
- Cooked mode crlf parameter controls whether to pass only a linefeed (NEWLINE) or both carriage return and a linefeed
- Cooked mode buffers and delivers an entire line
- Cbreak supports echo but doesn’t buffer (thus no line kill)

Driver Structure

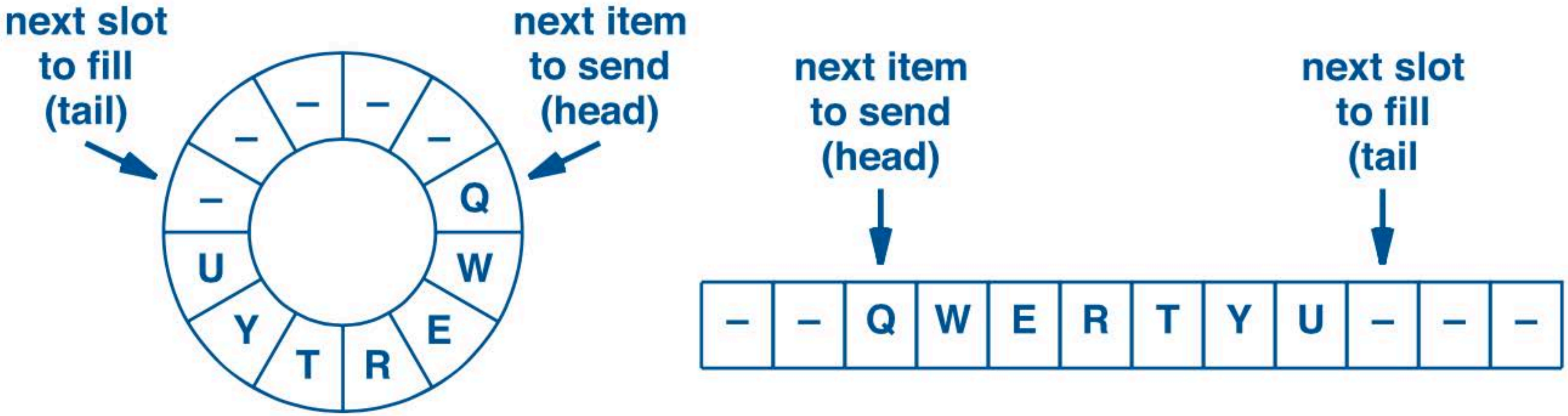
- Upper half (called by the application) and lower half (called when interrupts occur)
- Shared data structure between them
- Upper half copies data in and out of the data structure, to and from applications
- Lower half copies data in and out and delivers or retrieves from hardware
- Two halves are decoupled

Request Queue and Buffered I/O

- In general, there two items in a device driver's data structures
    - Queue of requests
    - The data itself
- The upper half places requests from applications into the request queue
    - Possibly breaking larger requests into sizes the device can handle
- Input and output data is generally buffered
    - Characters may be typed and received before an application has requested them
    - Applications can write larger amounts of data in a single operation
- Output buffering allows an application to read or write bytes when the device may operate on entire blocks

Tty Driver

- The tty driver uses circular input and output buffers
  - – Logically circular, implemented as an array



Moving Data Out

- Output functions copy data into the output buffer
  - Starting output interrupts on the device if it was idle
- When an output interrupt is generated, the lower half extracts bytes and puts them in the device's FIFO
- When the FIFO has been drained, the device interrupts again

Moving Data In

- When the device receives input, it interrupts, calling the lower-half handler function
  - ttyhandler()
- The handler copies data from the device’s FIFO into the input ring buffer
- When a process calls read(), the data is copied from the input ring buffer into process memory
- Block if no data available

Synchronization

- The upper and lower halves clearly need to synchronize
- Essentially, this is the producer/consumer problem and we know how to solve this with semaphores
- Input is straightforward – the upper half can wait() for input and the lower half can signal() when input is available
- Output is less obvious – recall that interrupt processing cannot cause the running process to block as it might be running in the idle process's context
  - So it cannot call wait()

Synchronization - Output

- The solution is to turn the problem around
- The lower half can be viewed as a producer of free slots
- So, the upper half can wait() for space in the output buffer and the lower half can signal() when space is available

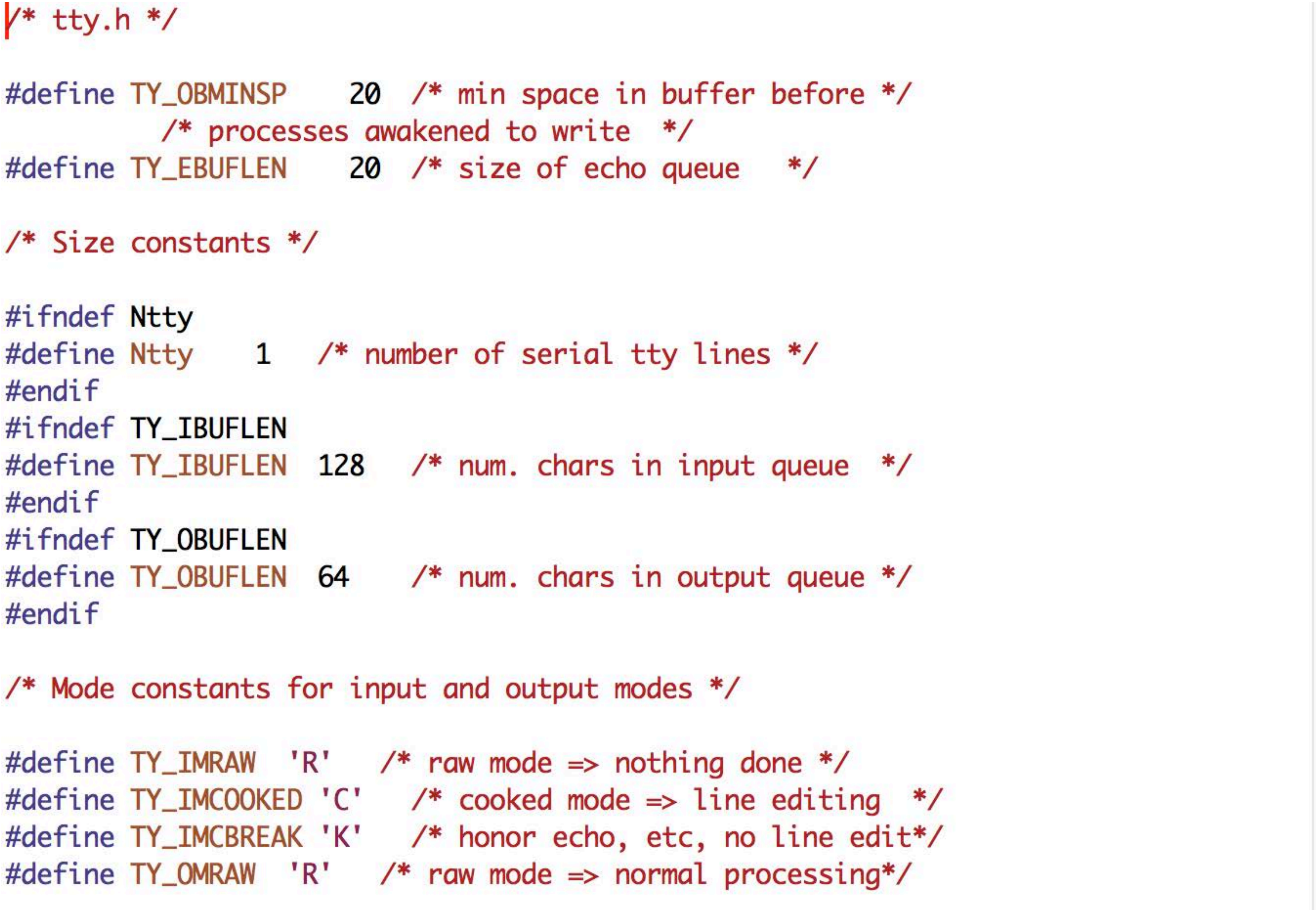
UART FIFOs

- Input and output FIFOs
- Most UARTs buffer more than one character
  - 16 in each direction in our case
- The device will interrupt when the first character arrives, but the FIFO will continue to fill
- Given potentially more than one character to be read, the hander can’t simply call signal()
  - Might result in immediate rescheduling
- The solution is to use resched_cntl to defer rescheduling until the FIFO has been emptied

Control Blocks

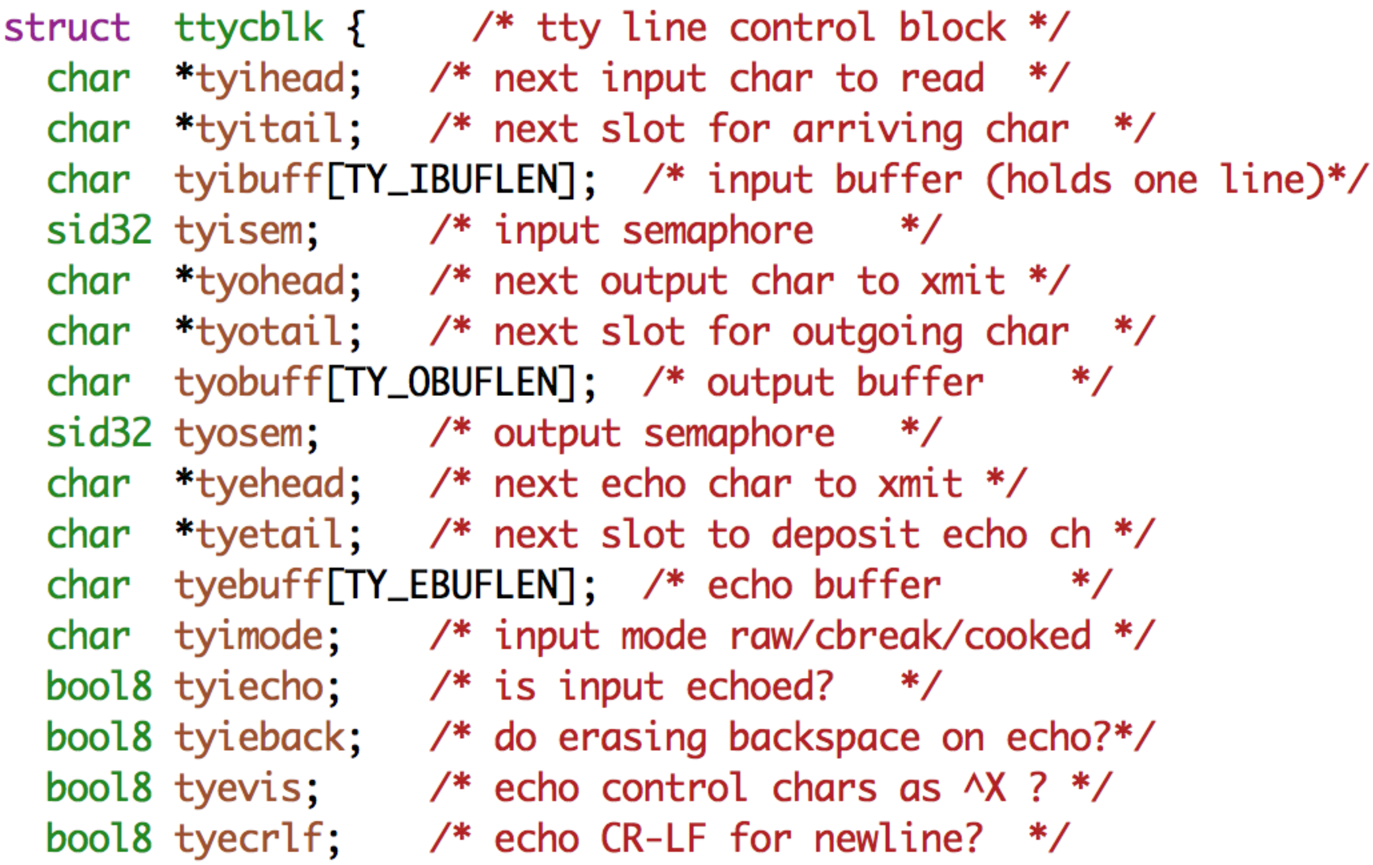
- Per instance data structure for control information
  - Semaphores, (pointers to) buffers, etc.
  - One copy of the driver
- Minor device number used as an index into the control block array
- Device driver functions get an argument indicating the specific control block
- The Tty device uses the ttycblk structure
  - input, output and echo buffers

tty.h



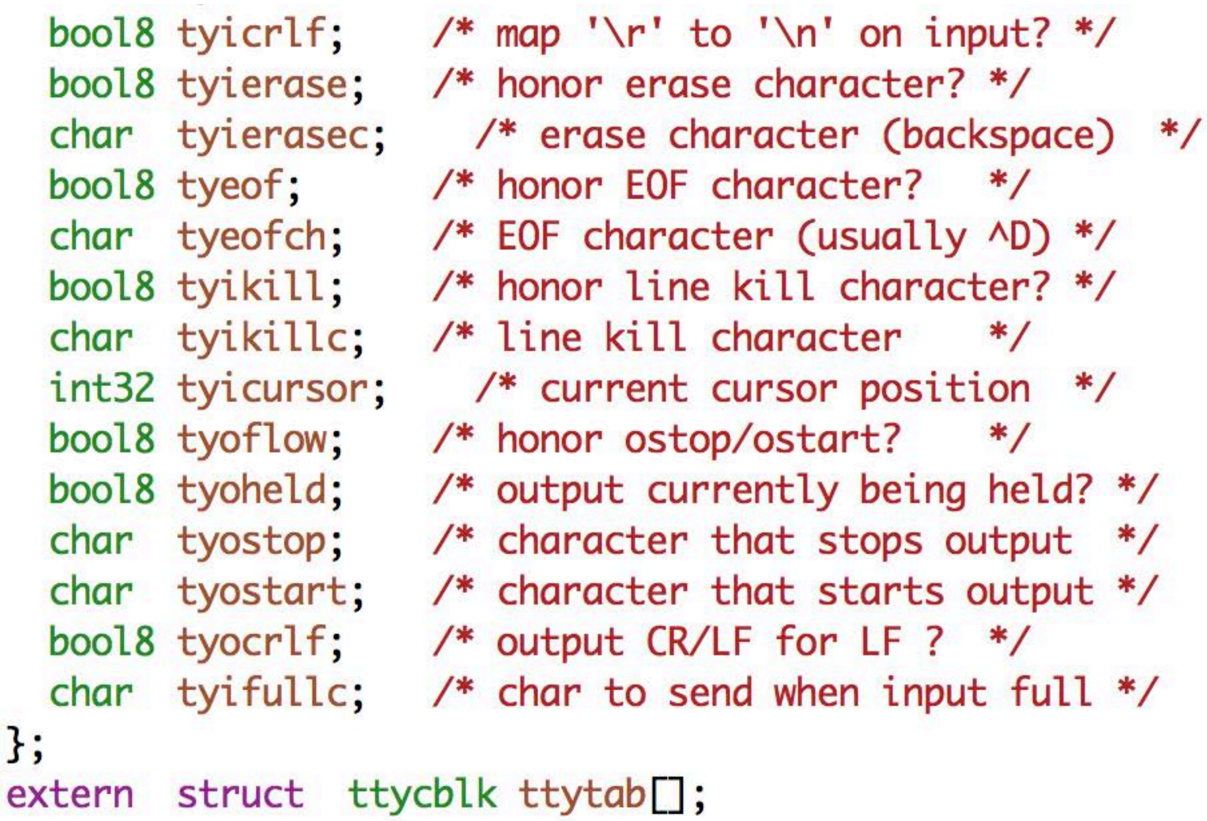
```
/* tty.h */

#define TY_OBMINSP    20  /* min space in buffer before */
          /* processes awakened to write  */
#define TY_EBUFLEN    20  /* size of echo queue   */

/* Size constants */

#ifndef Ntty
#define Ntty    1   /* number of serial tty lines */
#endif
#ifndef TY_IBUFLEN
#define TY_IBUFLEN  128   /* num. chars in input queue  */
#endif
#ifndef TY_OBUFLEN
#define TY_OBUFLEN  64    /* num. chars in output queue */
#endif

/* Mode constants for input and output modes */

#define TY_IMRAW   'R'   /* raw mode => nothing done */
#define TY_IMCOOKED 'C'   /* cooked mode => line editing  */
#define TY_IMCBREAK 'K'   /* honor echo, etc, no line edit*/
#define TY_OMRAW   'R'   /* raw mode => normal processing*/
```

tty.h



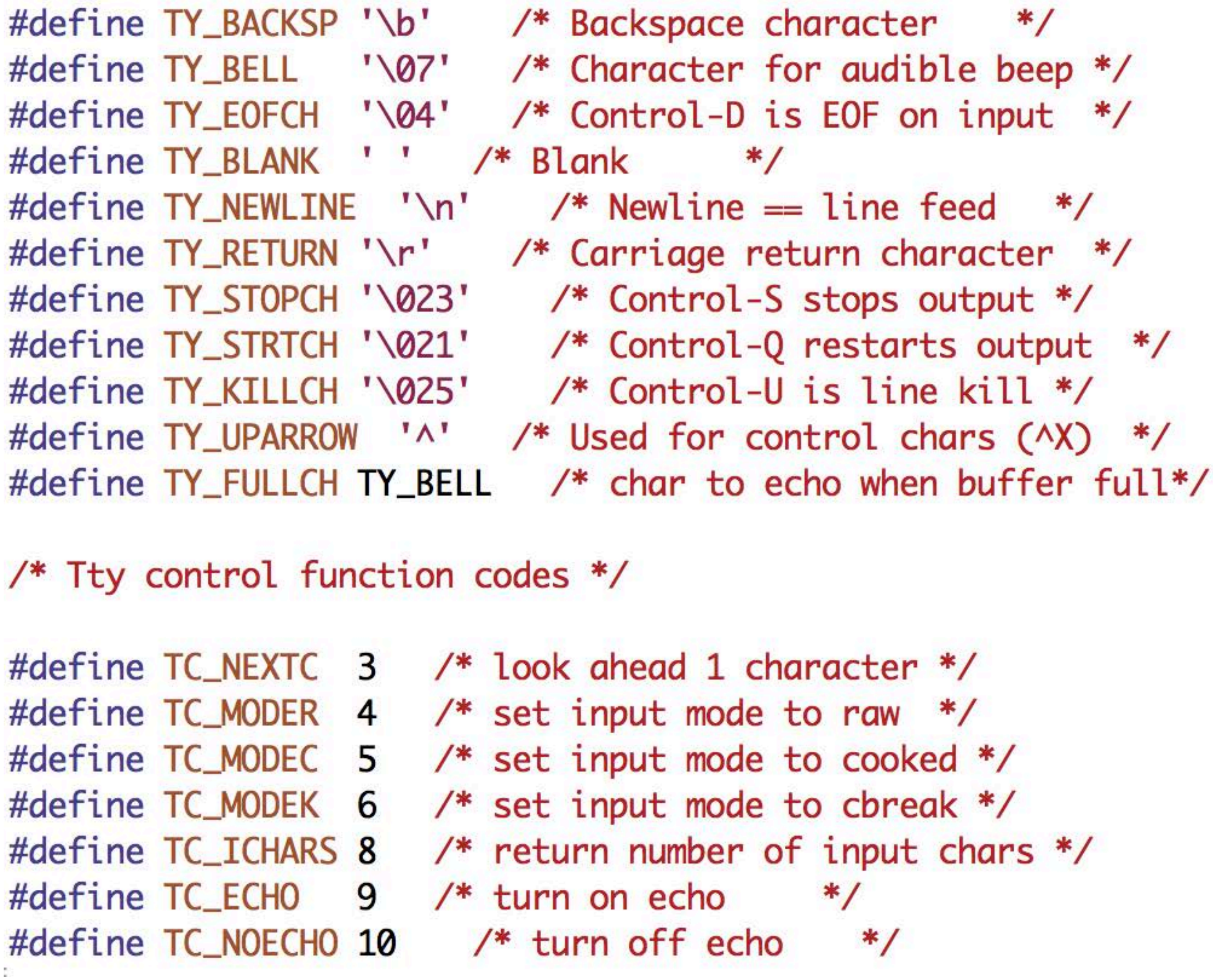
```
struct  ttycblk {      /* tty line control block */
  char  *tyihead;    /* next input char to read  */
  char  *tyitail;    /* next slot for arriving char  */
  char  tyibuff[TY_IBUFLEN];  /* input buffer (holds one line)*/
  sid32 tyisem;      /* input semaphore    */
  char  *tyohead;    /* next output char to xmit */
  char  *tyotail;    /* next slot for outgoing char  */
  char  tyobuff[TY_OBUFLEN];  /* output buffer    */
  sid32 tyosem;      /* output semaphore   */
  char  *tyehead;    /* next echo char to xmit */
  char  *tyetail;    /* next slot to deposit echo ch */
  char  tyebuff[TY_EBUFLEN];  /* echo buffer      */
  char  tyimode;     /* input mode raw/cbreak/cooked */
  bool8 tyiecho;     /* is input echoed?   */
  bool8 tyieback;    /* do erasing backspace on echo?*/
  bool8 tyevis;      /* echo control chars as ^X ? */
  bool8 tyecrlf;     /* echo CR-LF for newline?  */
```

tty.h



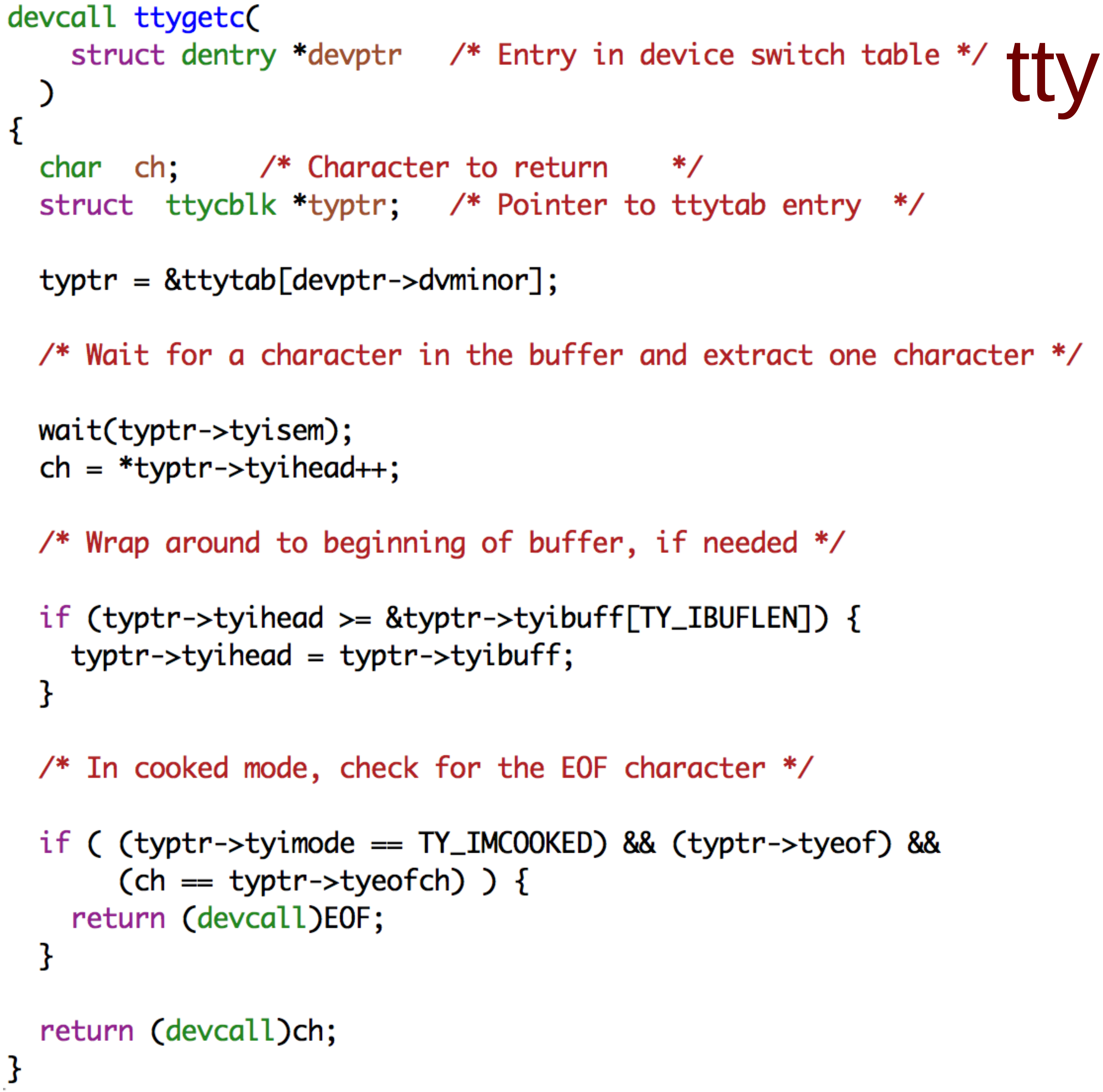
```
  bool8 tyicrlf;     /* map '\r' to '\n' on input? */
  bool8 tyierase;    /* honor erase character? */
  char  tyierasec;     /* erase character (backspace)  */
  bool8 tyeof;       /* honor EOF character?    */
  char  tyeofch;     /* EOF character (usually ^D) */
  bool8 tyikill;     /* honor line kill character? */
  char  tyikillc;    /* line kill character     */
  int32 tyicursor;     /* current cursor position  */
  bool8 tyoflow;     /* honor ostop/ostart?     */
  bool8 tyoheld;     /* output currently being held? */
  char  tyostop;     /* character that stops output  */
  char  tyostart;    /* character that starts output */
  bool8 tyocrlf;     /* output CR/LF for LF ?   */
  char  tyifullc;    /* char to send when input full */
};
extern  struct  ttycblk ttytab[];
```

tty.h



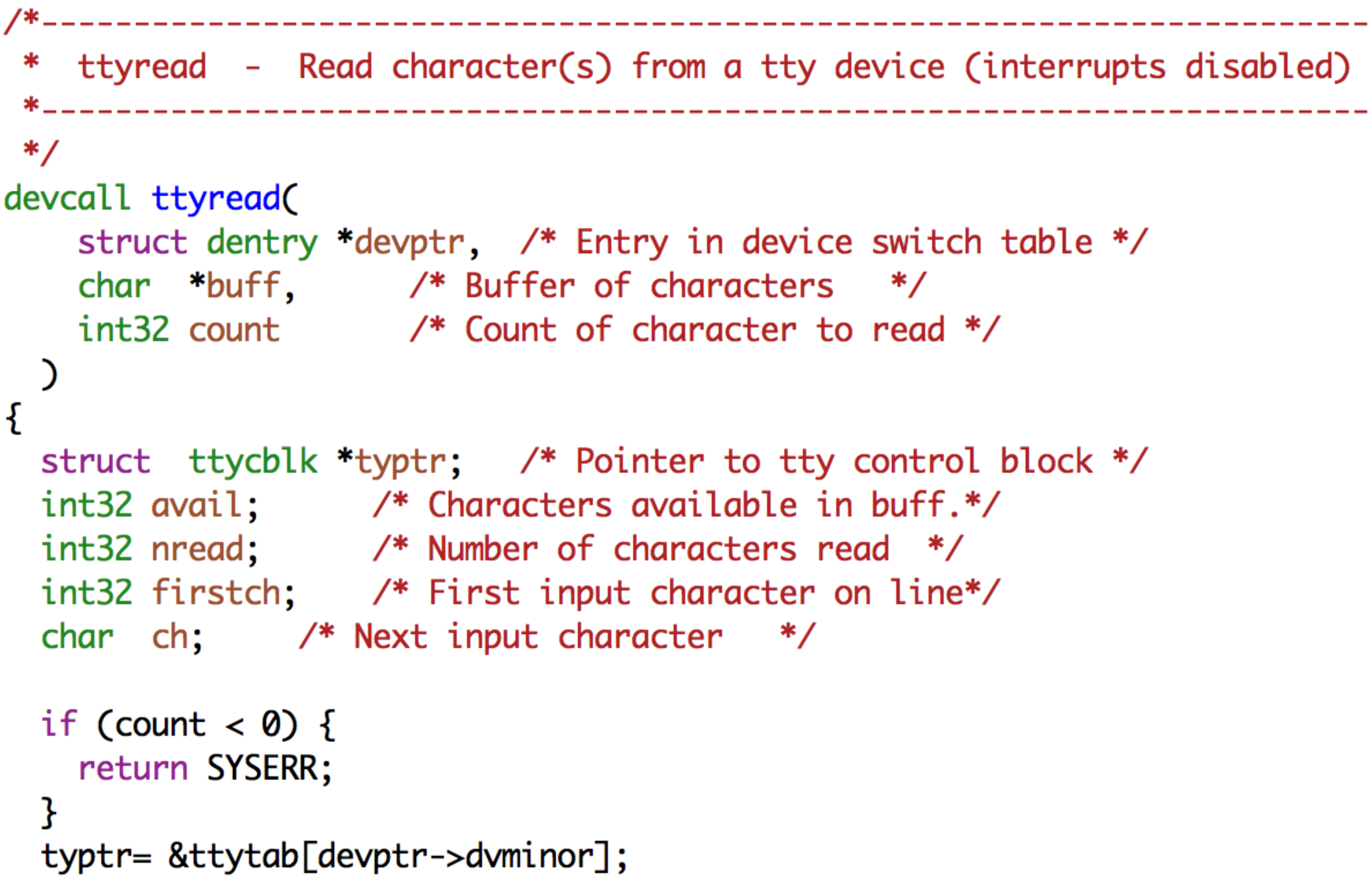
```
#define TY_BACKSP '\b'    /* Backspace character    */
#define TY_BELL   '\07'   /* Character for audible beep */
#define TY_EOFCH  '\04'   /* Control-D is EOF on input  */
#define TY_BLANK  ' '   /* Blank      */
#define TY_NEWLINE  '\n'    /* Newline == line feed   */
#define TY_RETURN '\r'    /* Carriage return character  */
#define TY_STOPCH '\023'    /* Control-S stops output */
#define TY_STRTCH '\021'    /* Control-Q restarts output  */
#define TY_KILLCH '\025'    /* Control-U is line kill */
#define TY_UPARROW  '^'   /* Used for control chars (^X)  */
#define TY_FULLCH TY_BELL   /* char to echo when buffer full*/

/* Tty control function codes */

#define TC_NEXTC  3   /* look ahead 1 character */
#define TC_MODER  4   /* set input mode to raw  */
#define TC_MODEC  5   /* set input mode to cooked */
#define TC_MODEK  6   /* set input mode to cbreak */
#define TC_ICHARS 8   /* return number of input chars */
#define TC_ECHO   9   /* turn on echo     */
#define TC_NOECHO 10    /* turn off echo    */
```

ttygetc.c



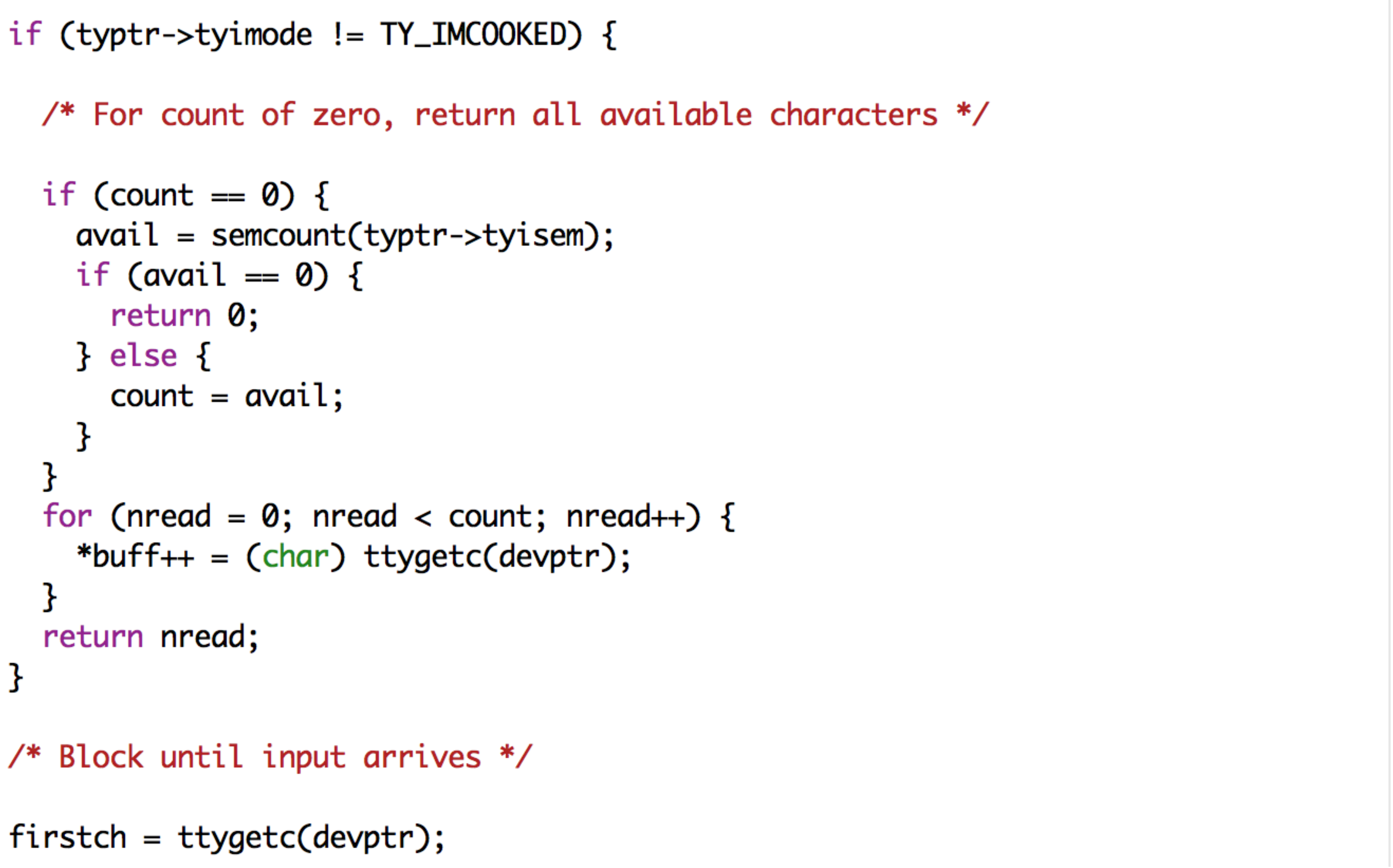
```
devcall ttygetc(
    struct dentry *devptr   /* Entry in device switch table */
  )
{
  char  ch;     /* Character to return    */
  struct  ttycblk *typtr;   /* Pointer to ttytab entry  */

  typtr = &ttytab[devptr->dvminor];

  /* Wait for a character in the buffer and extract one character */

  wait(typtr->tyisem);
  ch = *typtr->tyihead++;

  /* Wrap around to beginning of buffer, if needed */

  if (typtr->tyihead >= &typtr->tyibuff[TY_IBUFLEN]) {
    typtr->tyihead = typtr->tyibuff;
  }

  /* In cooked mode, check for the EOF character */

  if ( (typtr->tyimode == TY_IMCOOKED) && (typtr->tyeof) &&
       (ch == typtr->tyeofch) ) {
    return (devcall)EOF;
  }

  return (devcall)ch;
}
```

ttyread.c



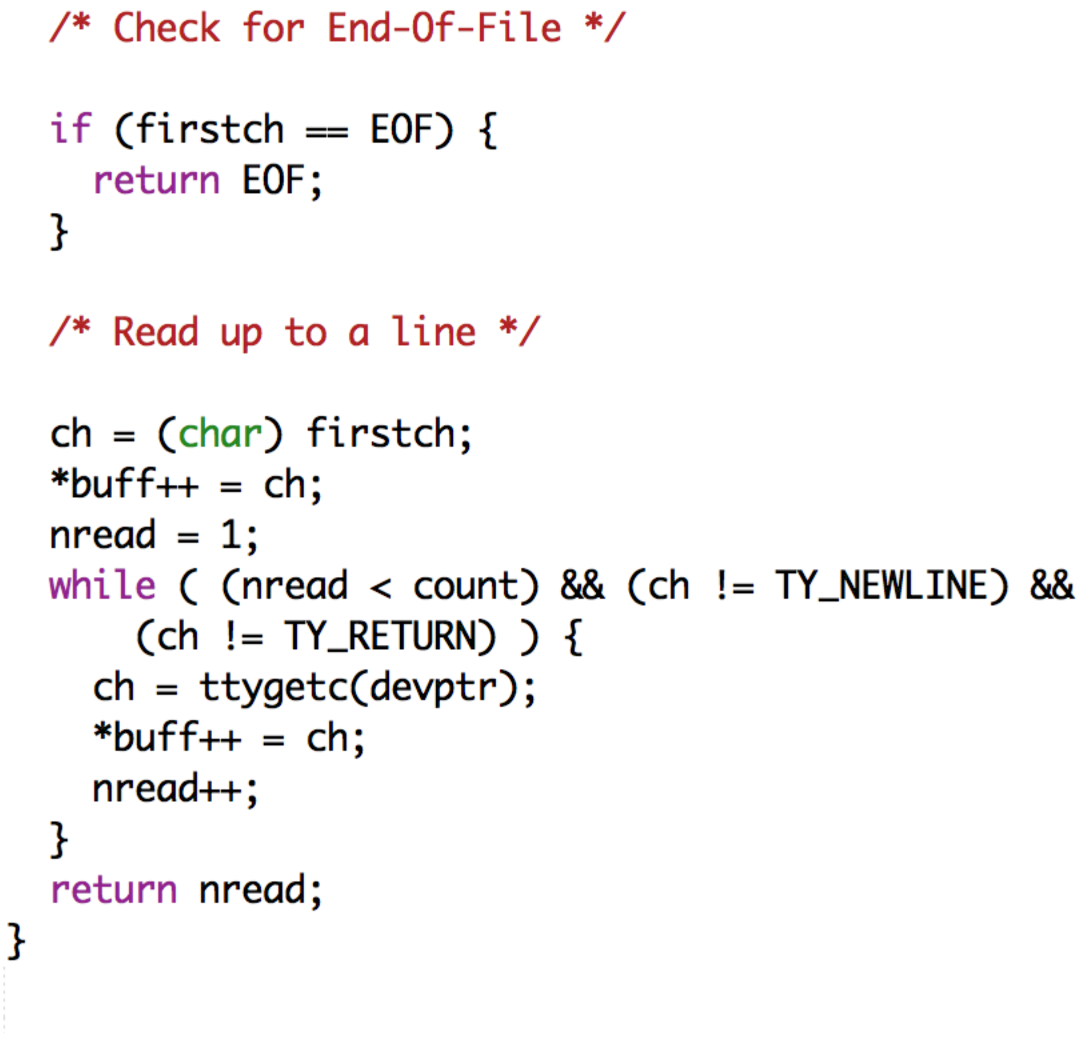
```
/*------------------------------------------------------------------------
 *  ttyread  -  Read character(s) from a tty device (interrupts disabled)
 *------------------------------------------------------------------------
 */
devcall ttyread(
    struct dentry *devptr,  /* Entry in device switch table */
    char  *buff,      /* Buffer of characters   */
    int32 count       /* Count of character to read */
  )
{
  struct  ttycblk *typtr;   /* Pointer to tty control block */
  int32 avail;      /* Characters available in buff.*/
  int32 nread;      /* Number of characters read  */
  int32 firstch;    /* First input character on line*/
  char  ch;     /* Next input character   */

  if (count < 0) {
    return SYSERR;
  }
  typtr= &ttytab[devptr->dvminor];
```

ttyread.c



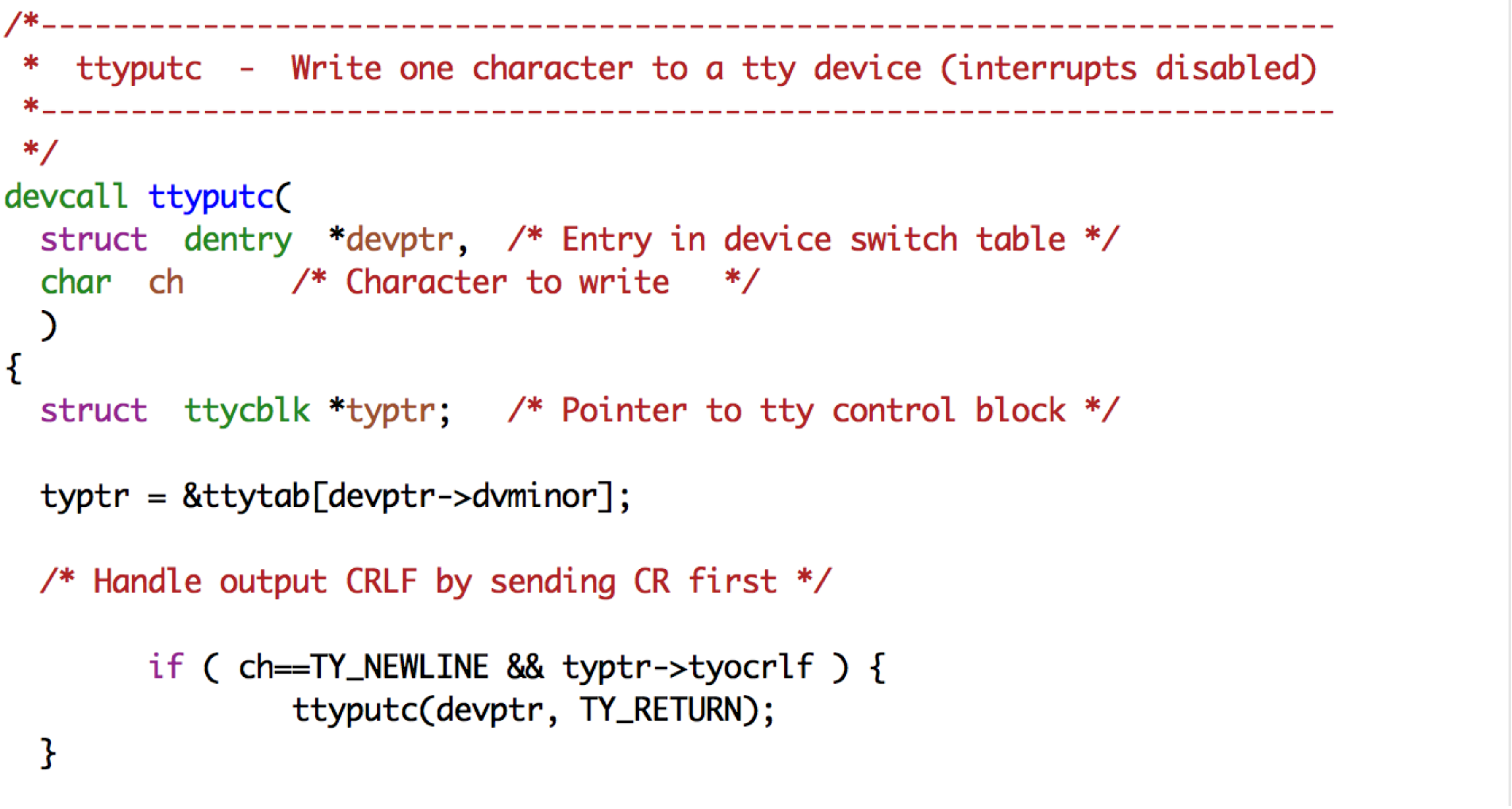
```
if (typtr->tyimode != TY_IMCOOKED) {

  /* For count of zero, return all available characters */

  if (count == 0) {
    avail = semcount(typtr->tyisem);
    if (avail == 0) {
      return 0;
    } else {
      count = avail;
    }
  }
  for (nread = 0; nread < count; nread++) {
    *buff++ = (char) ttygetc(devptr);
  }
  return nread;
}

/* Block until input arrives */

firstch = ttygetc(devptr);
```

ttyread.c



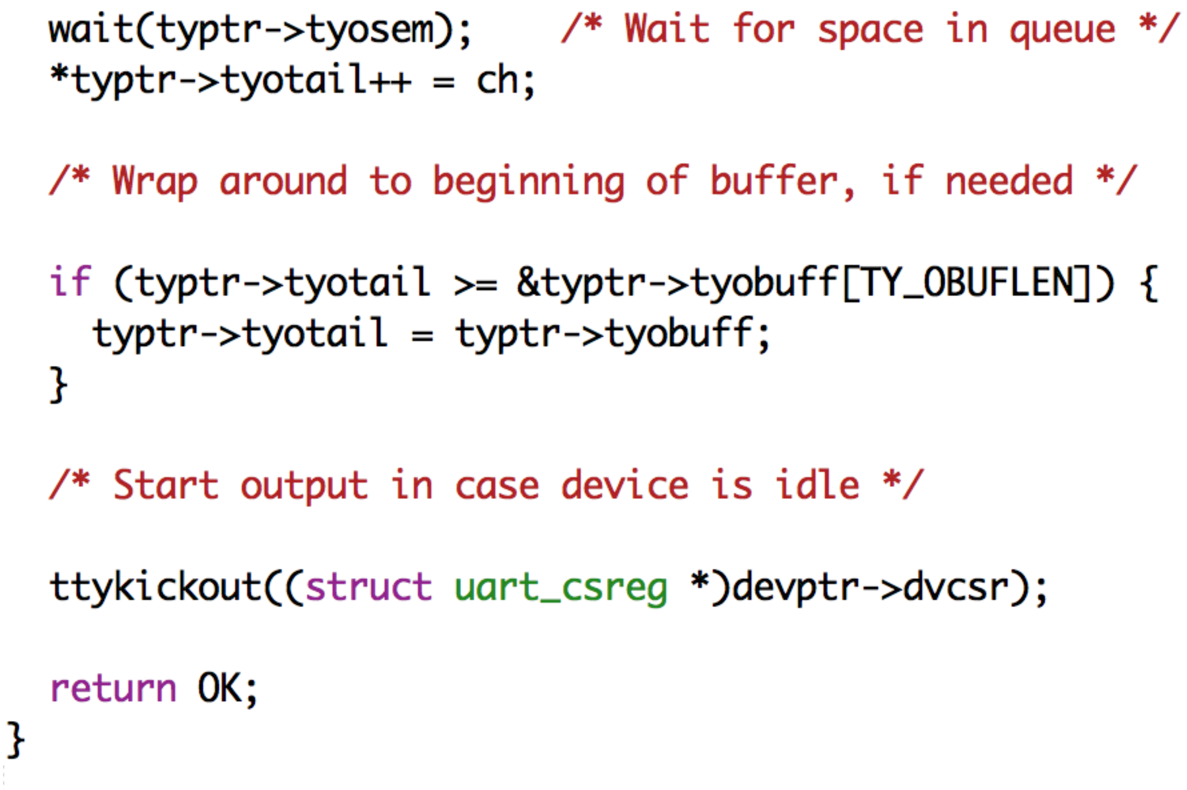
```c
  /* Check for End-Of-File */

  if (firstch == EOF) {
    return EOF;
  }

  /* Read up to a line */

  ch = (char) firstch;
  *buff++ = ch;
  nread = 1;
  while ( (nread < count) && (ch != TY_NEWLINE) &&
      (ch != TY_RETURN) ) {
    ch = ttygetc(devptr);
    *buff++ = ch;
    nread++;
  }
  return nread;
}
```

ttyputc.c



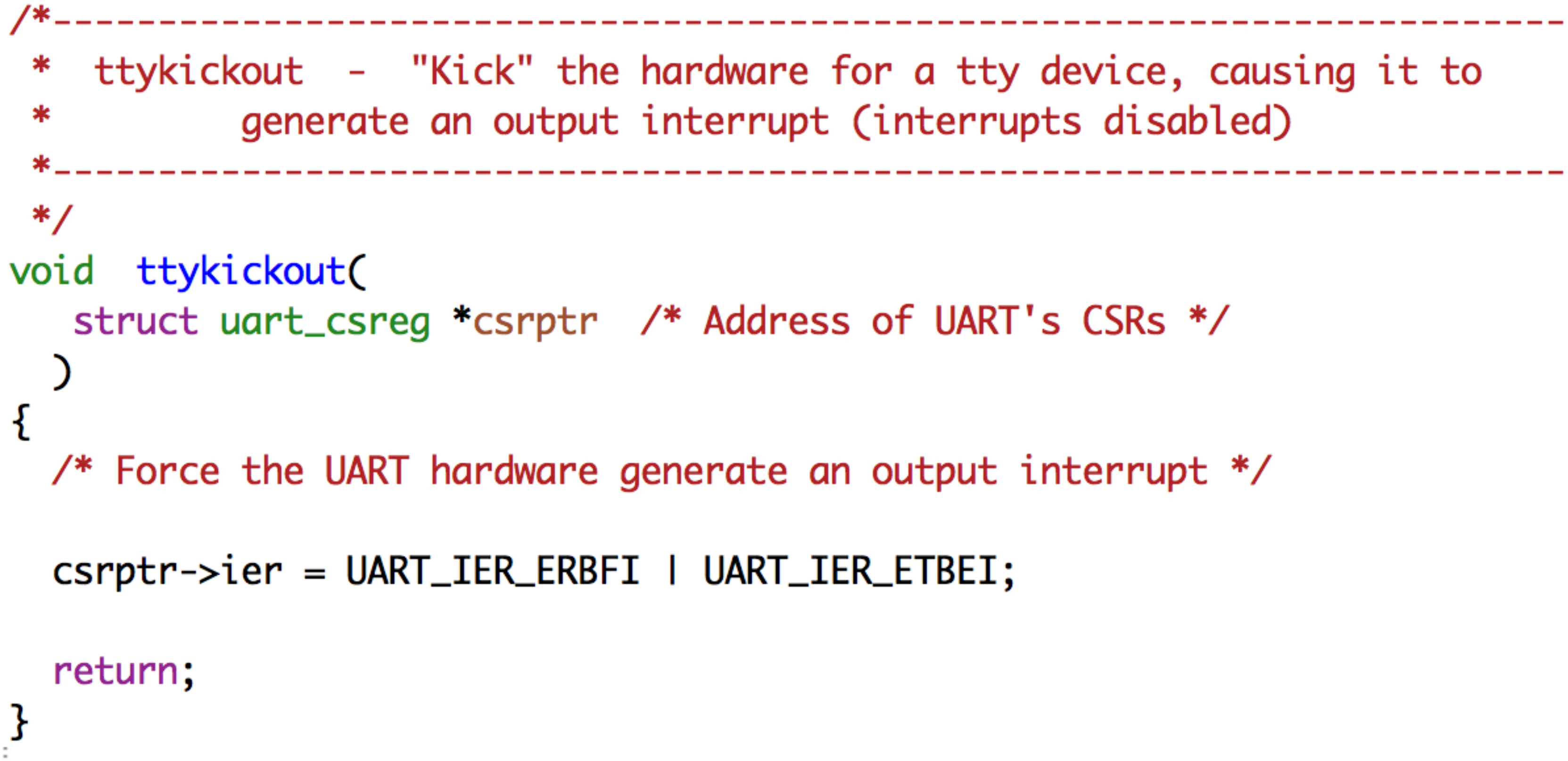
```
/*------------------------------------------------------------------------
 *  ttyputc  -  Write one character to a tty device (interrupts disabled)
 *------------------------------------------------------------------------
 */
devcall ttyputc(
  struct  dentry  *devptr,  /* Entry in device switch table */
  char    ch      /* Character to write   */
  )
{
  struct  ttycblk *typtr;   /* Pointer to tty control block */

  typtr = &ttytab[devptr->dvminor];

  /* Handle output CRLF by sending CR first */

        if ( ch==TY_NEWLINE && typtr->tyocrlf ) {
                ttyputc(devptr, TY_RETURN);
  }
```

ttyputc.c



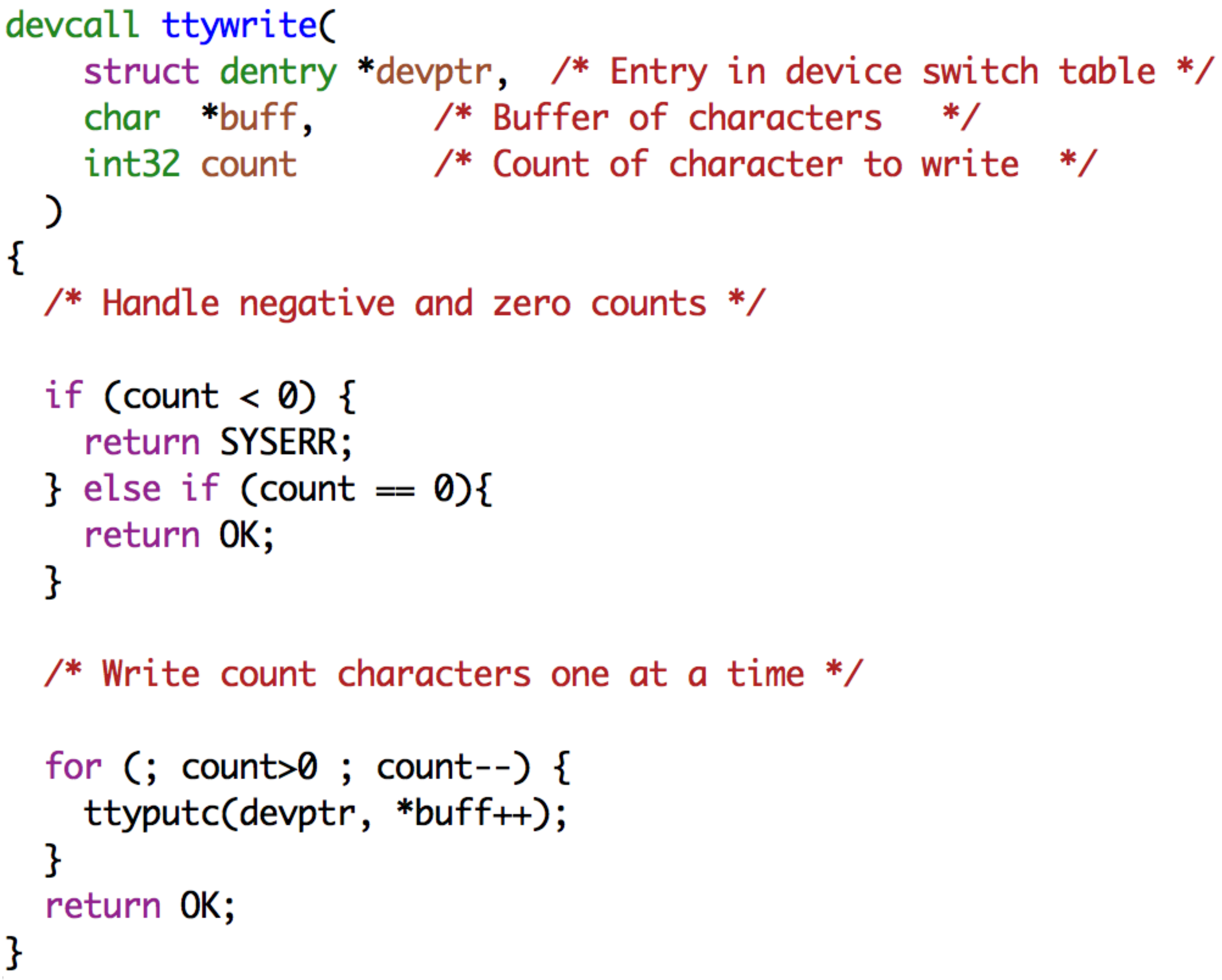
```
  wait(typtr->tyosem);    /* Wait for space in queue */
  *typtr->tyotail++ = ch;

  /* Wrap around to beginning of buffer, if needed */

  if (typtr->tyotail >= &typtr->tyobuff[TY_OBUFLEN]) {
    typtr->tyotail = typtr->tyobuff;
  }

  /* Start output in case device is idle */

  ttykickout((struct uart_csreg *)devptr->dvcsr);

  return OK;
}
```

ttykickout.c



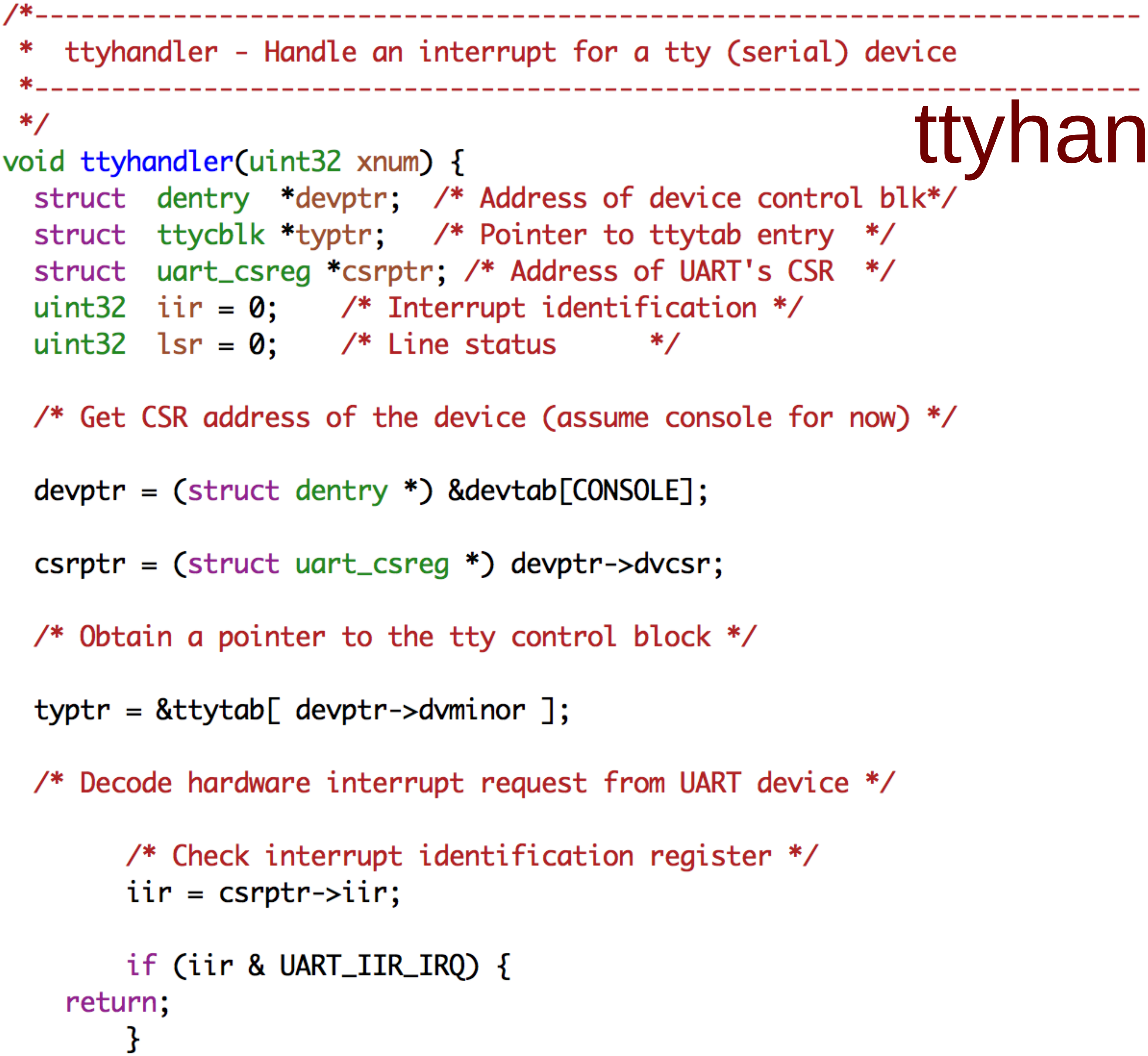
```
/*------------------------------------------------------------------------
 *  ttykickout  -  "Kick" the hardware for a tty device, causing it to
 *          generate an output interrupt (interrupts disabled)
 *------------------------------------------------------------------------
 */
void  ttykickout(
    struct uart_csreg *csrptr  /* Address of UART's CSRs */
   )
{
  /* Force the UART hardware generate an output interrupt */

  csrptr->ier = UART_IER_ERBFI | UART_IER_ETBEI;

  return;
}
```

ttywrite.c



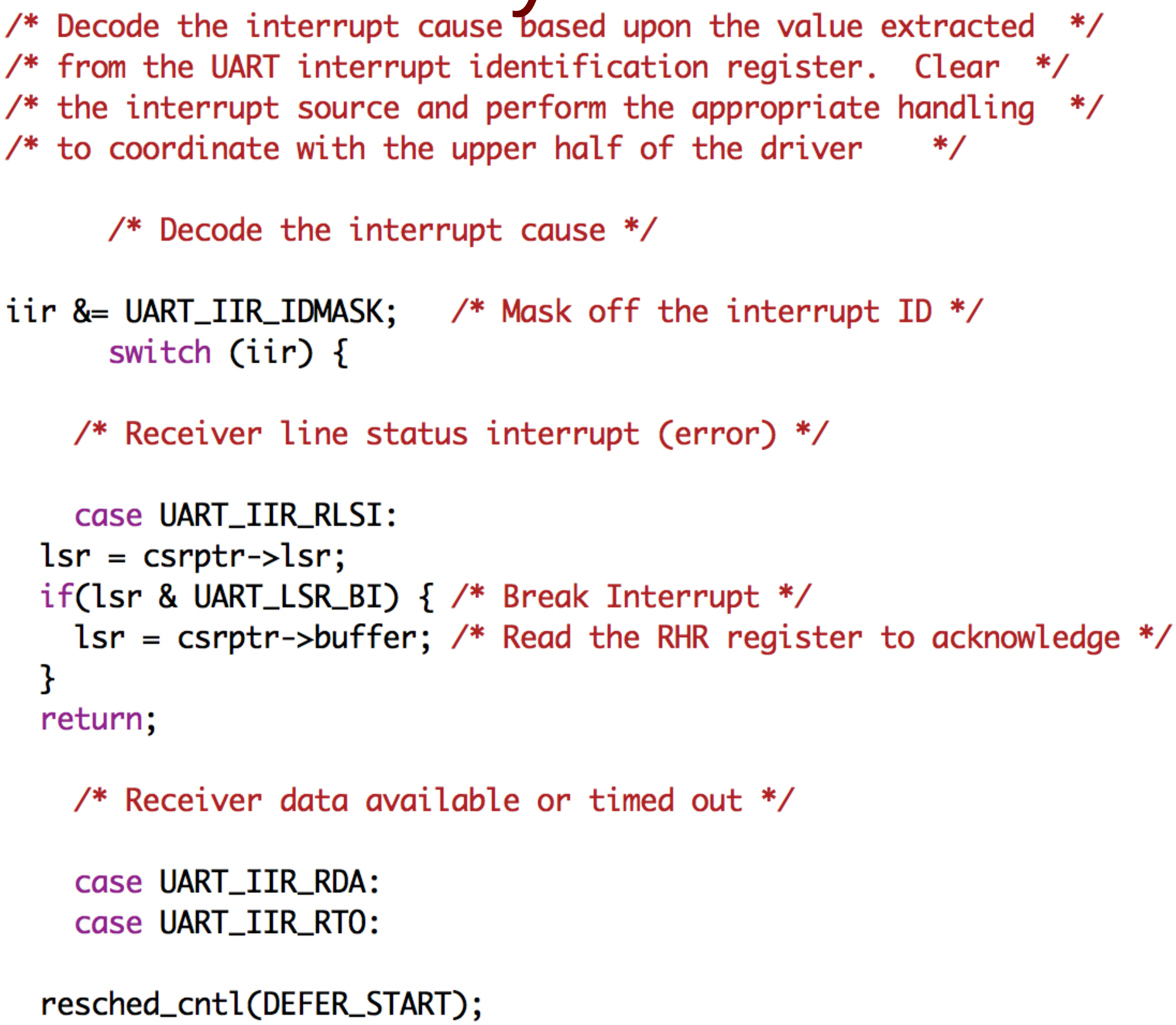
```
devcall ttywrite(
    struct dentry *devptr,  /* Entry in device switch table */
    char  *buff,      /* Buffer of characters   */
    int32 count       /* Count of character to write  */
  )
{
  /* Handle negative and zero counts */

  if (count < 0) {
    return SYSERR;
  } else if (count == 0){
    return OK;
  }

  /* Write count characters one at a time */

  for (; count>0 ; count--) {
    ttyputc(devptr, *buff++);
  }
  return OK;
}
```

ttyhandler.c



```
/*------------------------------------------------------------------------
 *  ttyhandler - Handle an interrupt for a tty (serial) device
 *------------------------------------------------------------------------
 */
void ttyhandler(uint32 xnum) {
  struct  dentry  *devptr;  /* Address of device control blk*/
  struct  ttycblk *typtr;   /* Pointer to ttytab entry  */
  struct  uart_csreg *csrptr; /* Address of UART's CSR  */
  uint32  iir = 0;     /* Interrupt identification */
  uint32  lsr = 0;     /* Line status      */

  /* Get CSR address of the device (assume console for now) */

  devptr = (struct dentry *) &devtab[CONSOLE];

  csrptr = (struct uart_csreg *) devptr->dvcsr;

  /* Obtain a pointer to the tty control block */

  typtr = &ttytab[ devptr->dvminor ];

  /* Decode hardware interrupt request from UART device */

        /* Check interrupt identification register */
        iir = csrptr->iir;

        if (iir & UART_IIR_IRQ) {
    return;
        }
```

ttyhandler.c

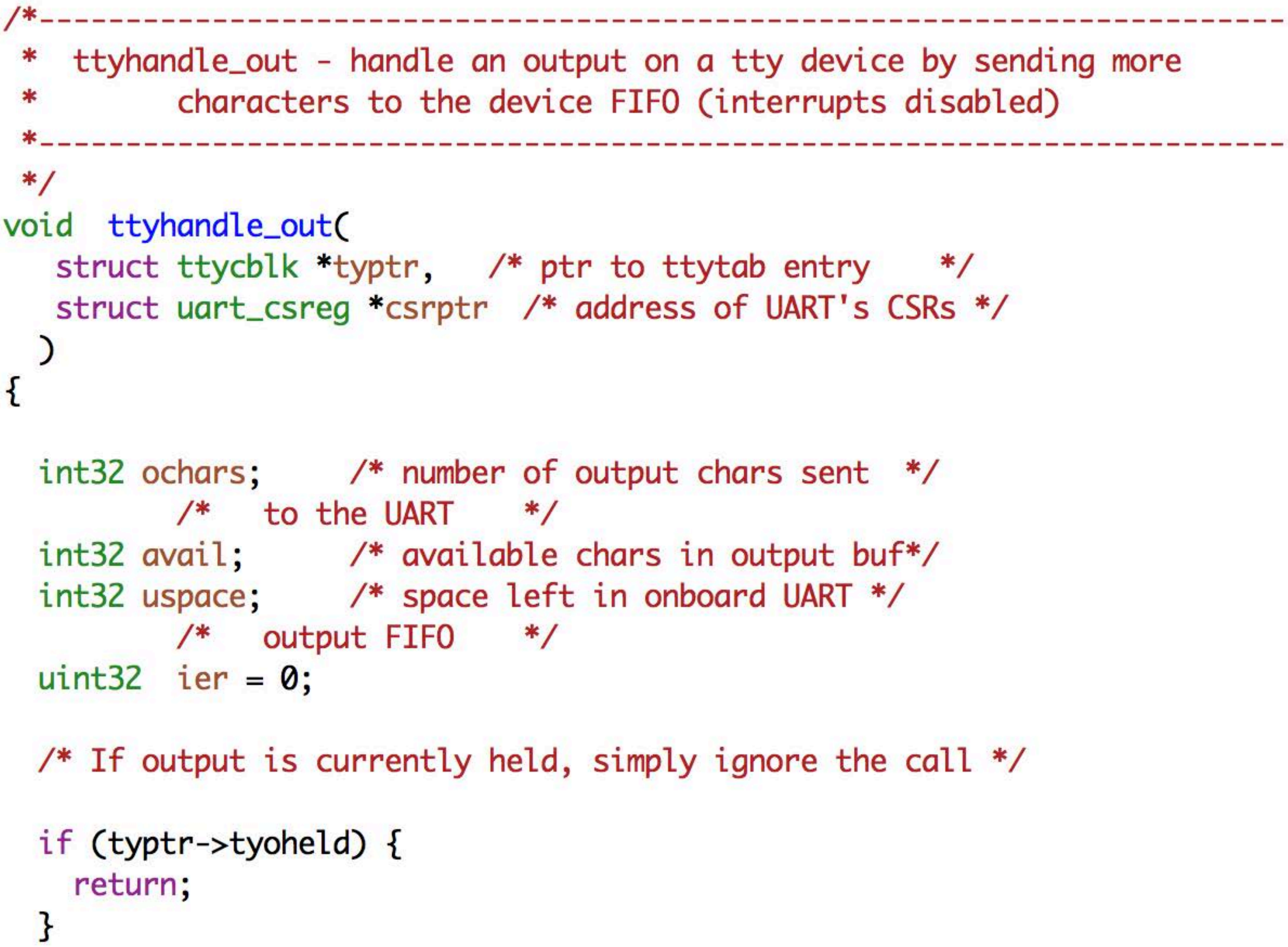


```
/* Decode the interrupt cause based upon the value extracted   */
/* from the UART interrupt identification register.  Clear   */
/* the interrupt source and perform the appropriate handling   */
/* to coordinate with the upper half of the driver    */

      /* Decode the interrupt cause */

iir &= UART_IIR_IDMASK;   /* Mask off the interrupt ID */
      switch (iir) {

    /* Receiver line status interrupt (error) */

    case UART_IIR_RLSI:
  lsr = csrptr->lsr;
  if(lsr & UART_LSR_BI) { /* Break Interrupt */
    lsr = csrptr->buffer; /* Read the RHR register to acknowledge */
  }
  return;

    /* Receiver data available or timed out */

    case UART_IIR_RDA:
    case UART_IIR_RTO:

  resched_cntl(DEFER_START);
```

ttyhandler.c

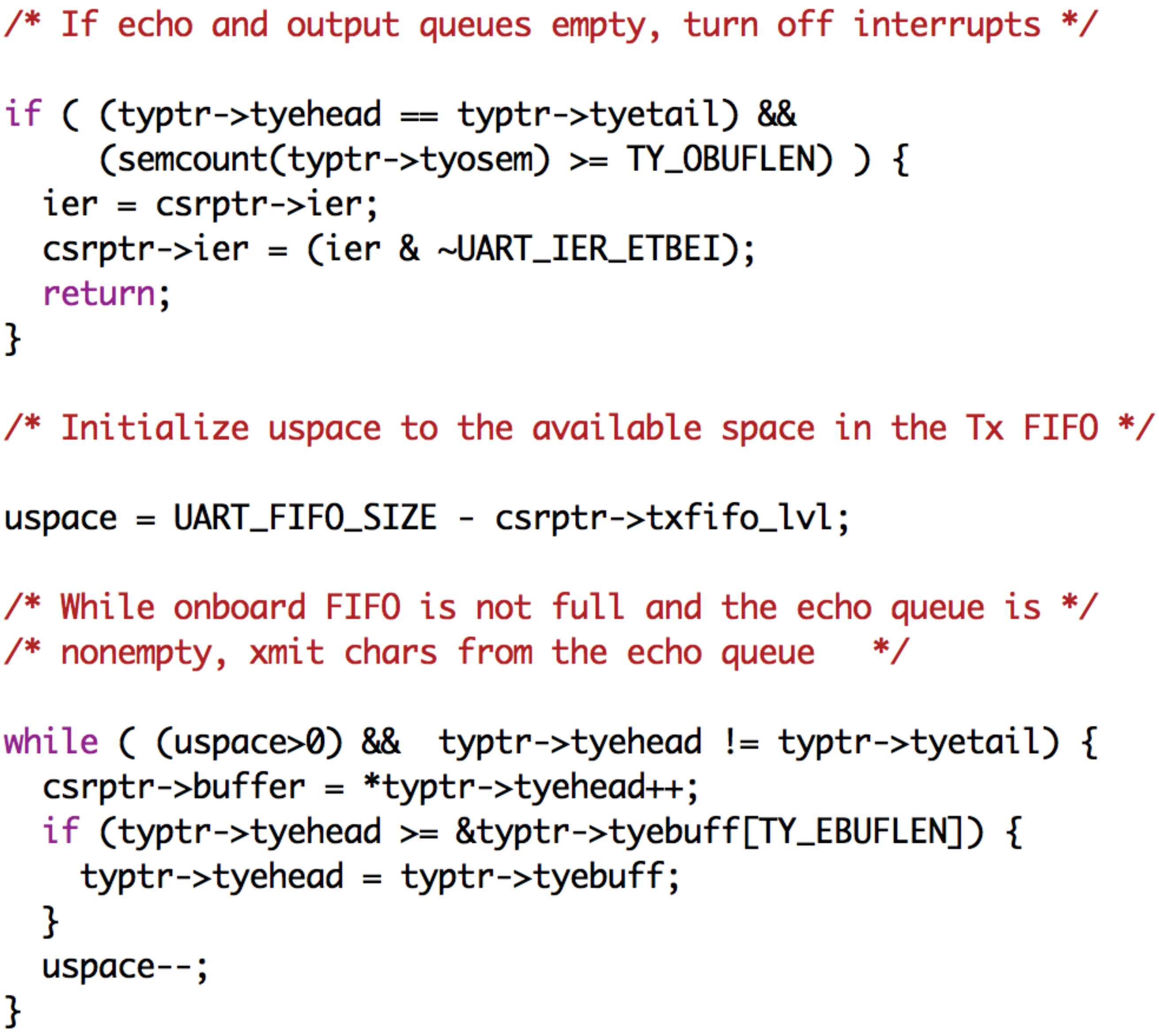
```
    /* While chars avail. in UART buffer, call ttyhandle_in */

    while ( (csrptr->lsr & UART_LSR_DR) != 0) {
      ttyhandle_in(typtr, csrptr);
                }

    resched_cntl(DEFER_STOP);

    return;

            /* Transmitter output FIFO is empty (i.e., ready for more)  */

      case UART_IIR_THRE:

    ttyhandle_out(typtr, csrptr);
    return;

      /* Modem status change (simply ignore) */

      case UART_IIR_MSC:
    return;
      }
}
```

ttyhandle_out.c



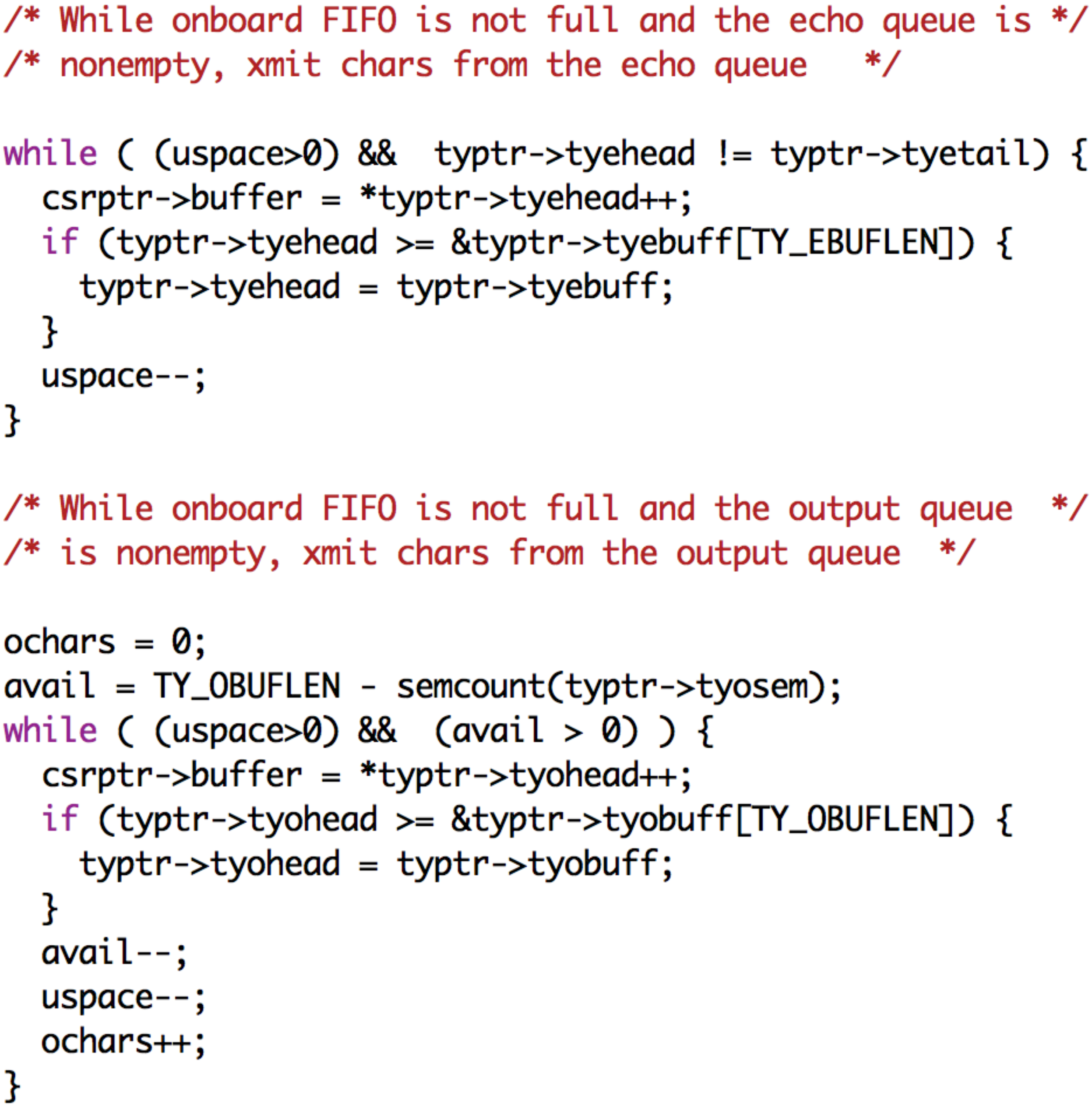
```
/*------------------------------------------------------------------------
 *  ttyhandle_out - handle an output on a tty device by sending more
 *        characters to the device FIFO (interrupts disabled)
 *------------------------------------------------------------------------
 */
void  ttyhandle_out(
   struct ttycblk *typtr,    /* ptr to ttytab entry    */
   struct uart_csreg *csrptr  /* address of UART's CSRs */
  )
{

  int32 ochars;      /* number of output chars sent  */
          /*    to the UART     */
  int32 avail;       /* available chars in output buf*/
  int32 uspace;      /* space left in onboard UART */
          /*    output FIFO     */
  uint32  ier = 0;

  /* If output is currently held, simply ignore the call */

  if (typtr->tyoheld) {
    return;
  }
```

ttyhandle_out.c



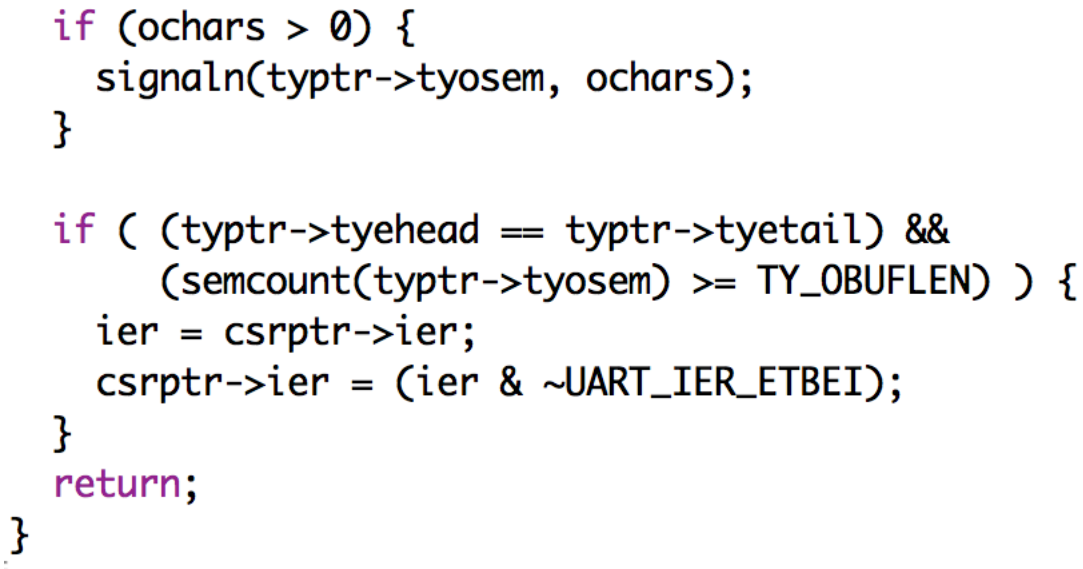
```
/* If echo and output queues empty, turn off interrupts */

if ( (typtr->tyehead == typtr->tyetail) &&
     (semcount(typtr->tyosem) >= TY_OBUFLEN) ) {
  ier = csrptr->ier;
  csrptr->ier = (ier & ~UART_IER_ETBEI);
  return;
}

/* Initialize uspace to the available space in the Tx FIFO */

uspace = UART_FIFO_SIZE - csrptr->txfifo_lvl;

/* While onboard FIFO is not full and the echo queue is */
/* nonempty, xmit chars from the echo queue    */

while ( (uspace>0) &&  typtr->tyehead != typtr->tyetail) {
  csrptr->buffer = *typtr->tyehead++;
  if (typtr->tyehead >= &typtr->tyebuff[TY_EBUFLEN]) {
    typtr->tyehead = typtr->tyebuff;
  }
  uspace--;
}
```

ttyhandle out.c



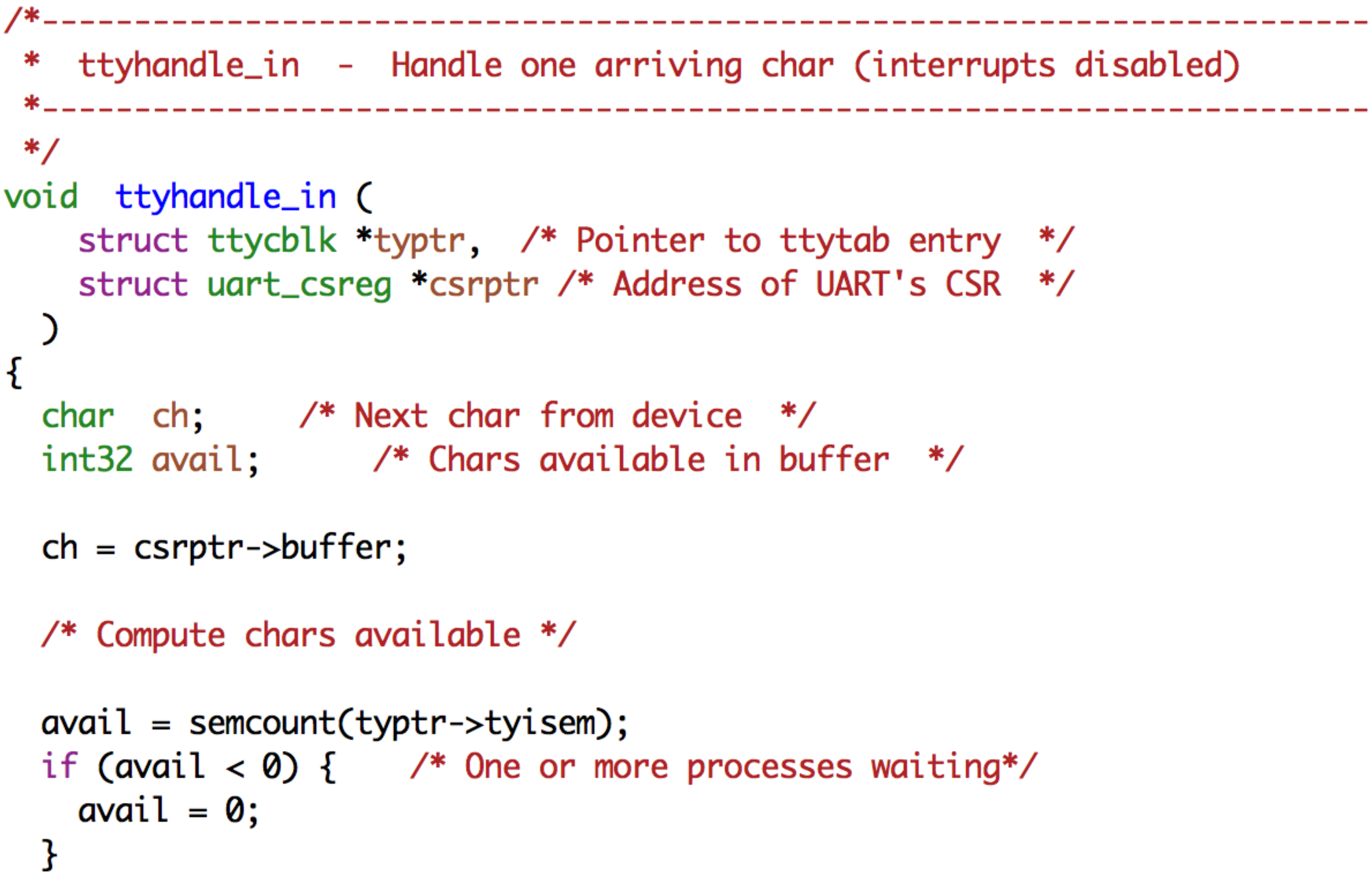
```
/* While onboard FIFO is not full and the echo queue is */
/* nonempty, xmit chars from the echo queue    */

while ( (uspace>0) &&  typtr->tyehead != typtr->tyetail) {
  csrptr->buffer = *typtr->tyehead++;
  if (typtr->tyehead >= &typtr->tyebuff[TY_EBUFLEN]) {
    typtr->tyehead = typtr->tyebuff;
  }
  uspace--;
}

/* While onboard FIFO is not full and the output queue  */
/* is nonempty, xmit chars from the output queue  */

ochars = 0;
avail = TY_OBUFLEN - semcount(typtr->tyosem);
while ( (uspace>0) &&  (avail > 0) ) {
  csrptr->buffer = *typtr->tyohead++;
  if (typtr->tyohead >= &typtr->tyobuff[TY_OBUFLEN]) {
    typtr->tyohead = typtr->tyobuff;
  }
  avail--;
  uspace--;
  ochars++;
}
```

ttyhandle_out.c



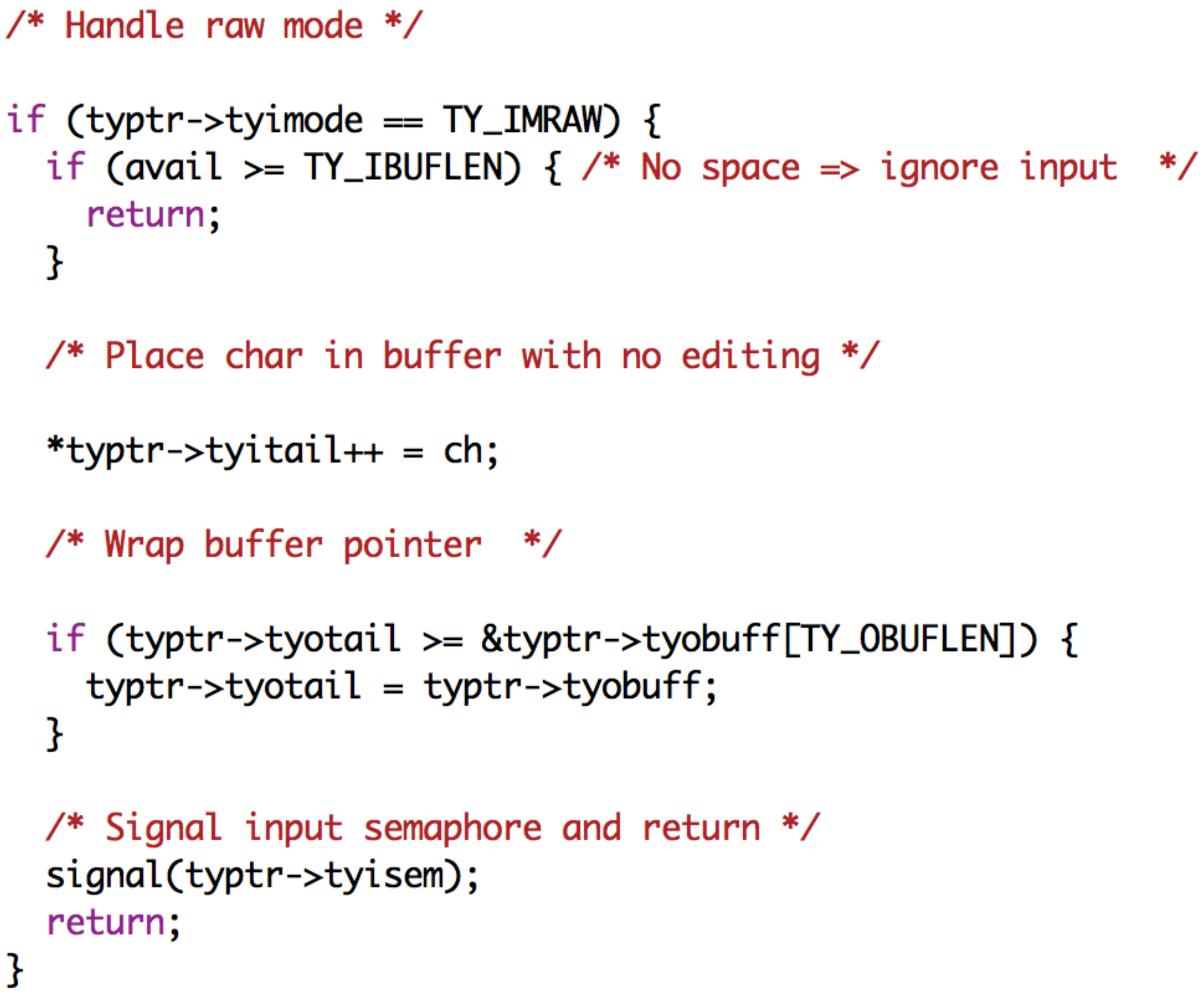
```
  if (ochars > 0) {
    signaln(typtr->tyosem, ochars);
  }

  if ( (typtr->tyehead == typtr->tyetail) &&
       (semcount(typtr->tyosem) >= TY_OBUFLEN) ) {
    ier = csrptr->ier;
    csrptr->ier = (ier & ~UART_IER_ETBEI);
  }
  return;
}
```

ttyhandle_in.c



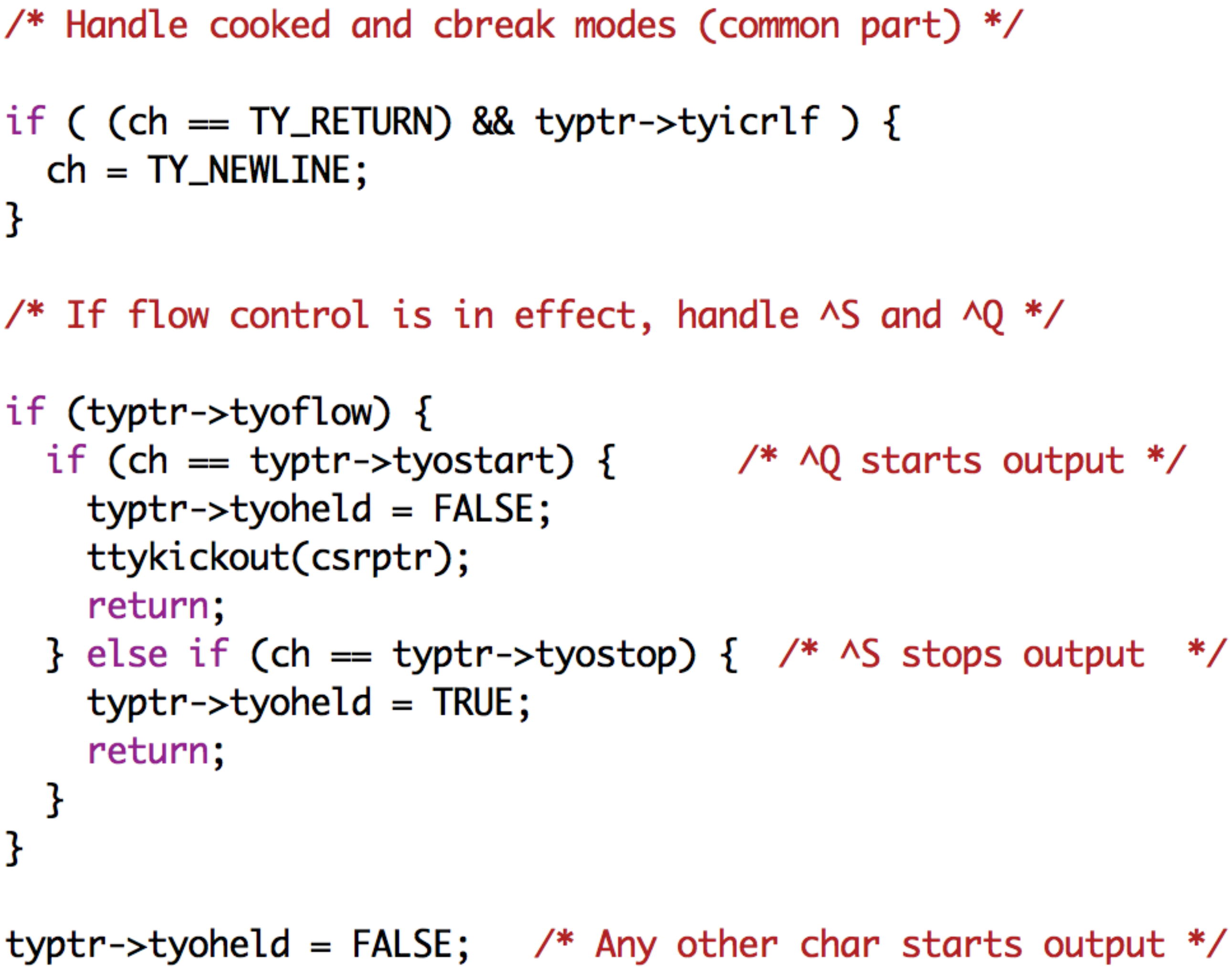
```c
/*------------------------------------------------------------------------
 *  ttyhandle_in  -  Handle one arriving char (interrupts disabled)
 *------------------------------------------------------------------------
 */
void  ttyhandle_in (
	struct ttycblk *typtr,	/* Pointer to ttytab entry	*/
	struct uart_csreg *csrptr /* Address of UART's CSR	*/
  )
{
  char  ch;		/* Next char from device	*/
  int32 avail;		/* Chars available in buffer	*/

  ch = csrptr->buffer;

  /* Compute chars available */

  avail = semcount(typtr->tyisem);
  if (avail < 0) {	/* One or more processes waiting*/
	avail = 0;
  }
```

ttyhandle_in.c



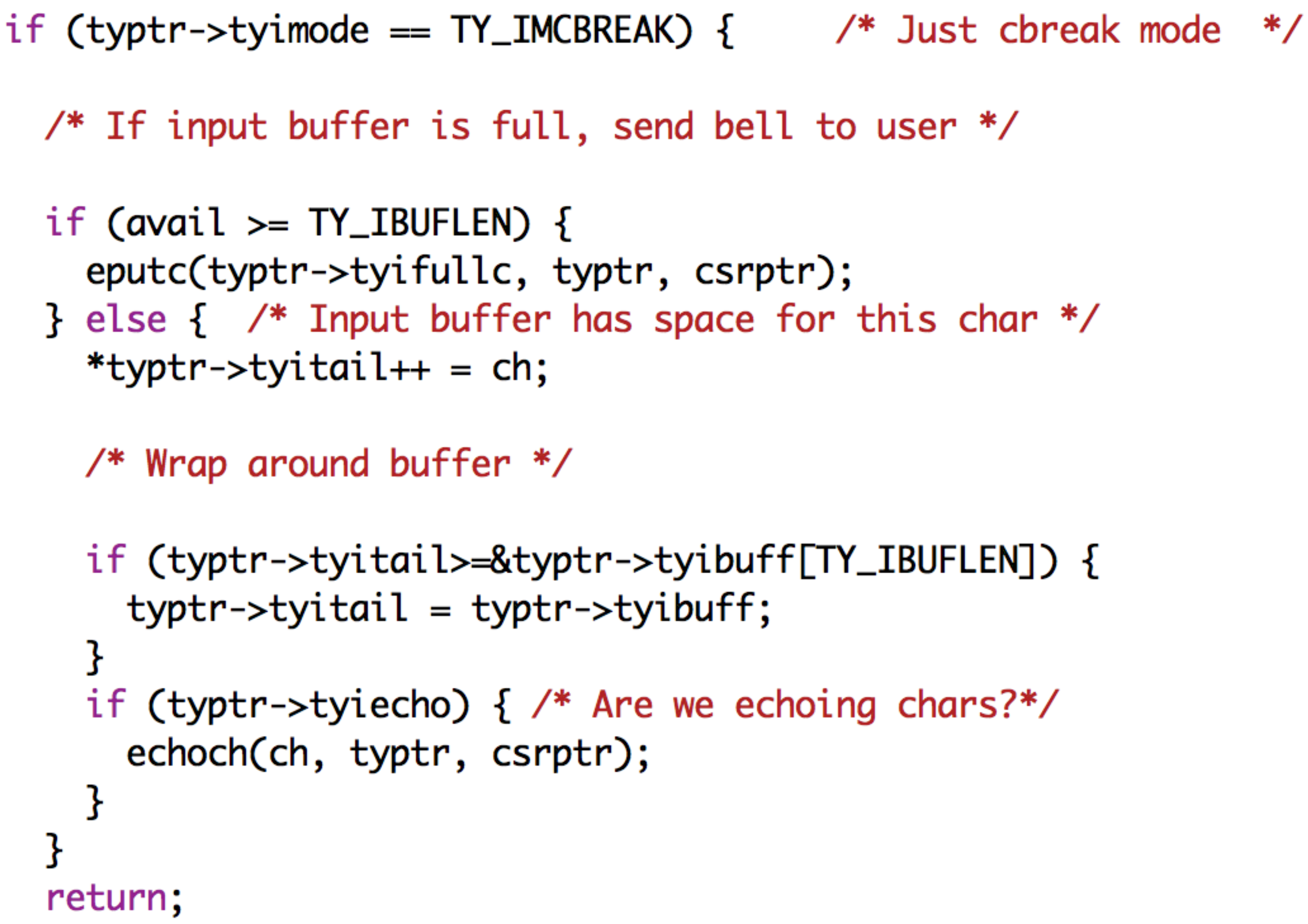
```
/* Handle raw mode */

if (typtr->tyimode == TY_IMRAW) {
  if (avail >= TY_IBUFLEN) { /* No space => ignore input  */
    return;
  }

  /* Place char in buffer with no editing */

  *typtr->tyitail++ = ch;

  /* Wrap buffer pointer  */

  if (typtr->tyotail >= &typtr->tyobuff[TY_OBUFLEN]) {
    typtr->tyotail = typtr->tyobuff;
  }

  /* Signal input semaphore and return */
  signal(typtr->tyisem);
  return;
}
```

ttyhandle_in.c



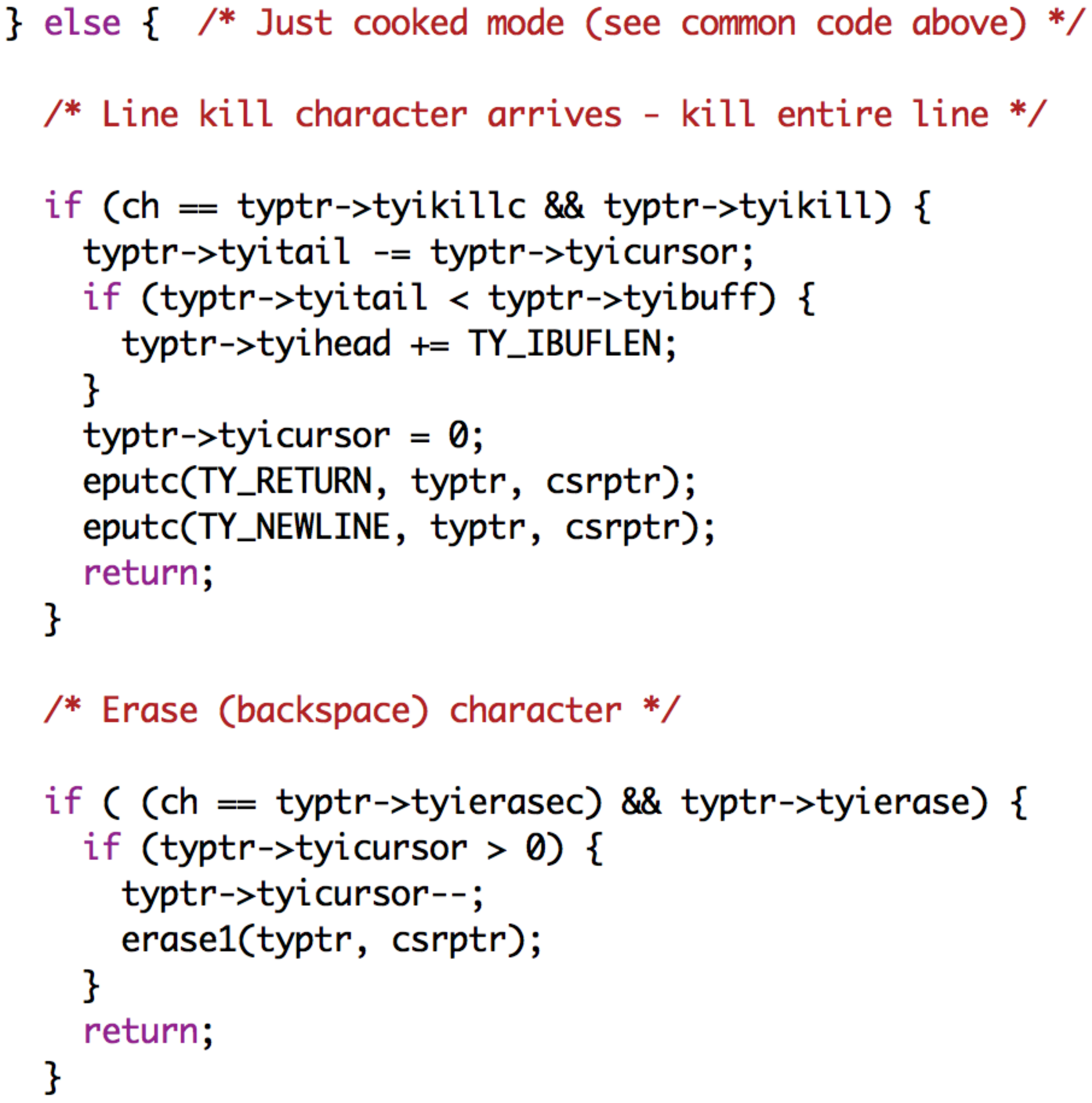
```
/* Handle cooked and cbreak modes (common part) */

if ( (ch == TY_RETURN) && typtr->tyicrlf ) {
  ch = TY_NEWLINE;
}

/* If flow control is in effect, handle ^S and ^Q */

if (typtr->tyoflow) {
  if (ch == typtr->tyostart) {      /* ^Q starts output */
    typtr->tyoheld = FALSE;
    ttykickout(csrptr);
    return;
  } else if (ch == typtr->tyostop) {  /* ^S stops output  */
    typtr->tyoheld = TRUE;
    return;
  }
}

typtr->tyoheld = FALSE;   /* Any other char starts output */
```

ttyhandle_in.c



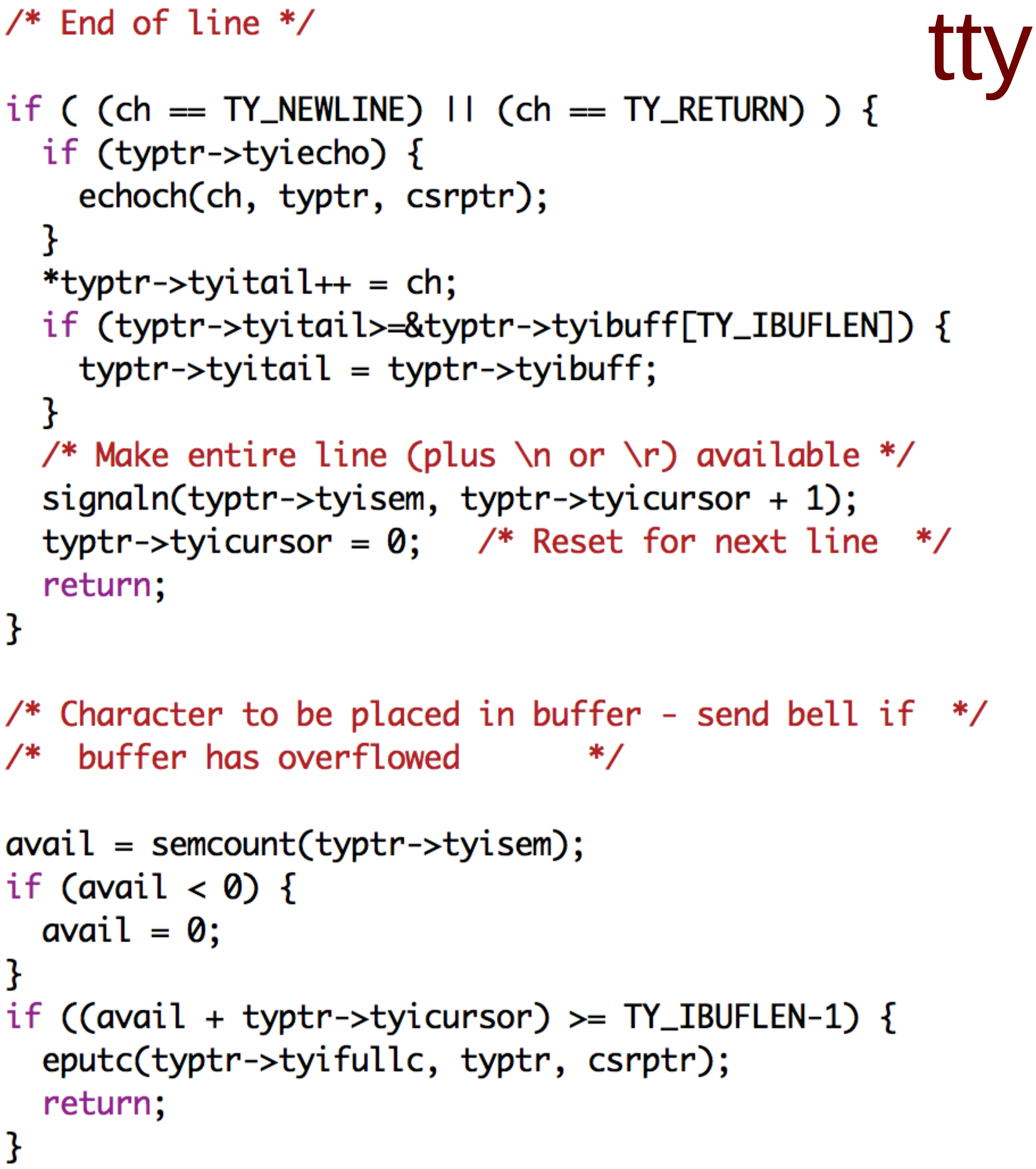
```c
if (typtr->tyimode == TY_IMCBREAK) {    /* Just cbreak mode  */

  /* If input buffer is full, send bell to user */

  if (avail >= TY_IBUFLEN) {
    eputc(typtr->tyifullc, typtr, csrptr);
  } else {  /* Input buffer has space for this char */
    *typtr->tyitail++ = ch;

    /* Wrap around buffer */

    if (typtr->tyitail>=&typtr->tyibuff[TY_IBUFLEN]) {
      typtr->tyitail = typtr->tyibuff;
    }
    if (typtr->tyiecho) { /* Are we echoing chars?*/
      echoch(ch, typtr, csrptr);
    }
  }
  return;
```

ttyhandle in.c



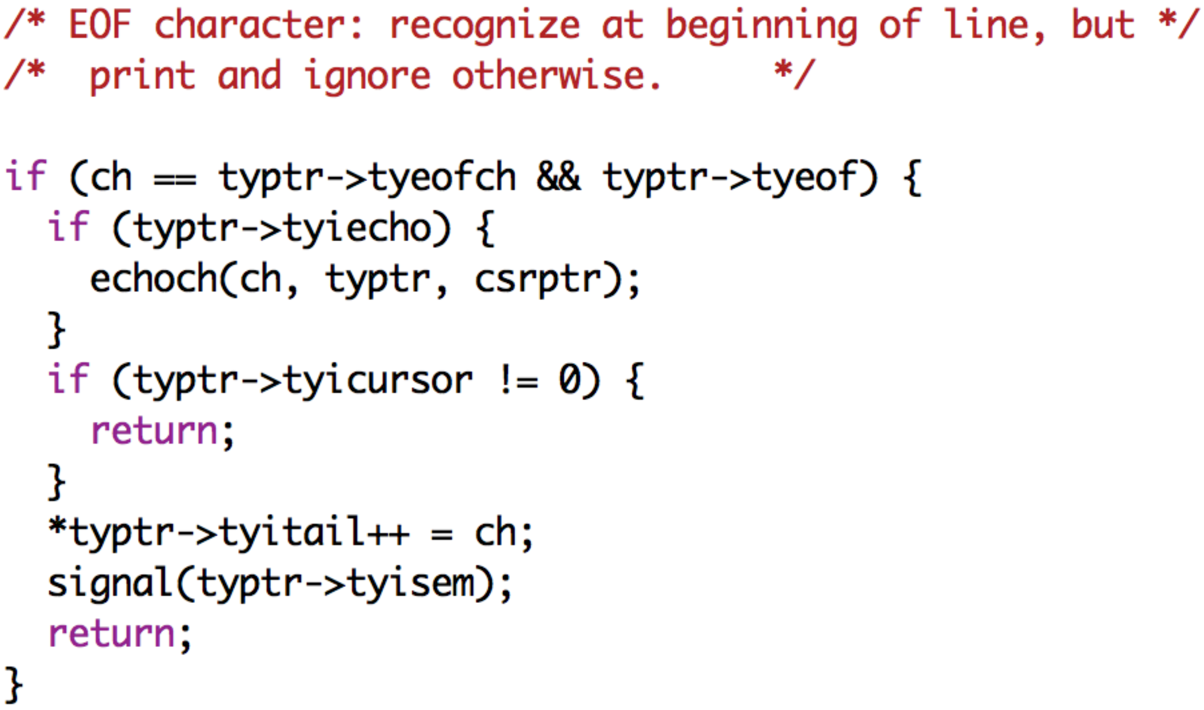
```
} else {  /* Just cooked mode (see common code above) */

  /* Line kill character arrives - kill entire line */

  if (ch == typtr->tyikillc && typtr->tyikill) {
    typtr->tyitail -= typtr->tyicursor;
    if (typtr->tyitail < typtr->tyibuff) {
      typtr->tyihead += TY_IBUFLEN;
    }
    typtr->tyicursor = 0;
    eputc(TY_RETURN, typtr, csrptr);
    eputc(TY_NEWLINE, typtr, csrptr);
    return;
  }

  /* Erase (backspace) character */

  if ( (ch == typtr->tyierasec) && typtr->tyierase) {
    if (typtr->tyicursor > 0) {
      typtr->tyicursor--;
      erase1(typtr, csrptr);
    }
    return;
  }
```

ttyhandle_in.c



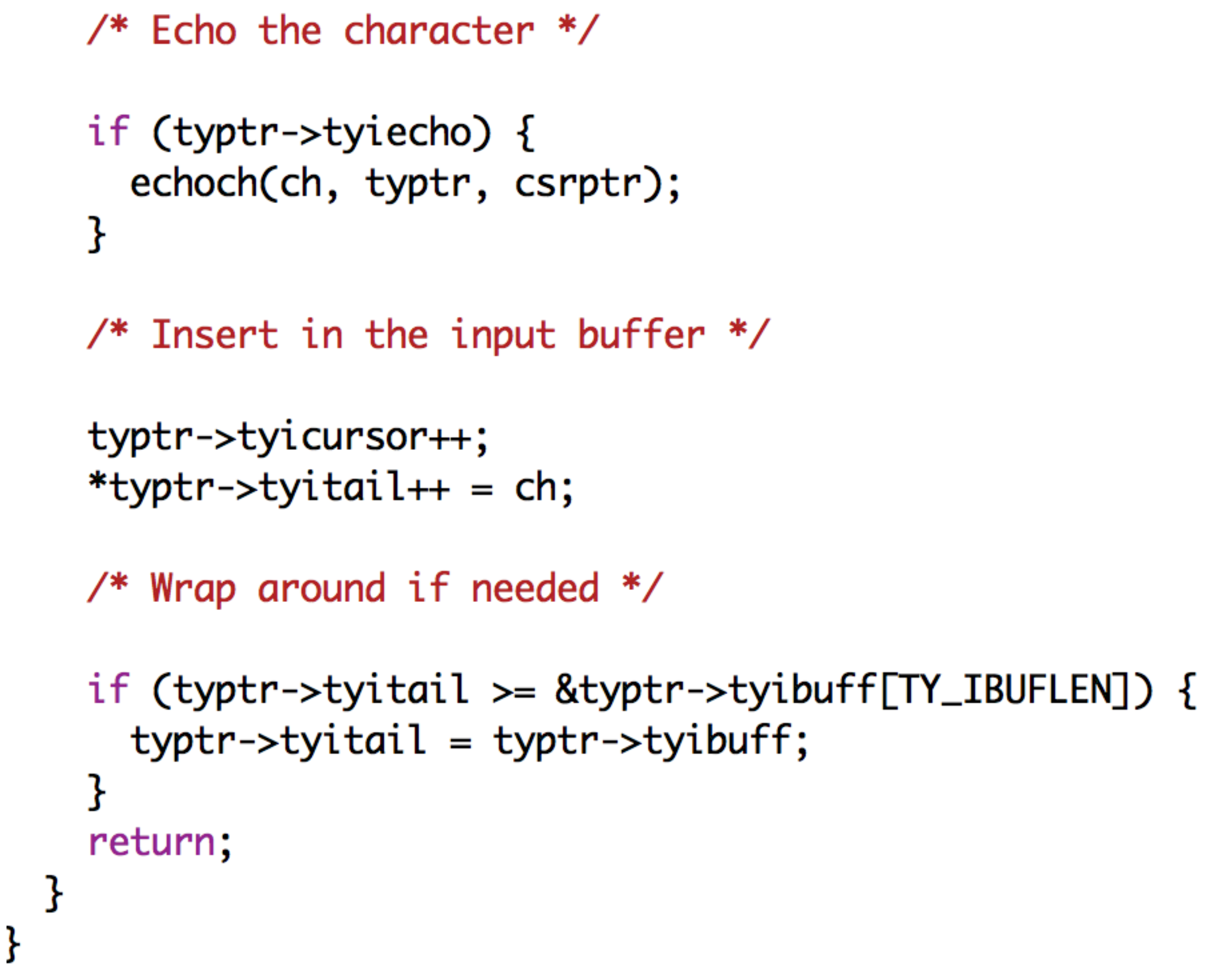
```
/* End of line */

if ( (ch == TY_NEWLINE) || (ch == TY_RETURN) ) {
  if (typtr->tyiecho) {
    echoch(ch, typtr, csrptr);
  }
  *typtr->tyitail++ = ch;
  if (typtr->tyitail>=&typtr->tyibuff[TY_IBUFLEN]) {
    typtr->tyitail = typtr->tyibuff;
  }
  /* Make entire line (plus \n or \r) available */
  signaln(typtr->tyisem, typtr->tyicursor + 1);
  typtr->tyicursor = 0;   /* Reset for next line  */
  return;
}

/* Character to be placed in buffer - send bell if  */
/*  buffer has overflowed       */

avail = semcount(typtr->tyisem);
if (avail < 0) {
  avail = 0;
}
if ((avail + typtr->tyicursor) >= TY_IBUFLEN-1) {
  eputc(typtr->tyifullc, typtr, csrptr);
  return;
}
```

ttyhandle_in.c



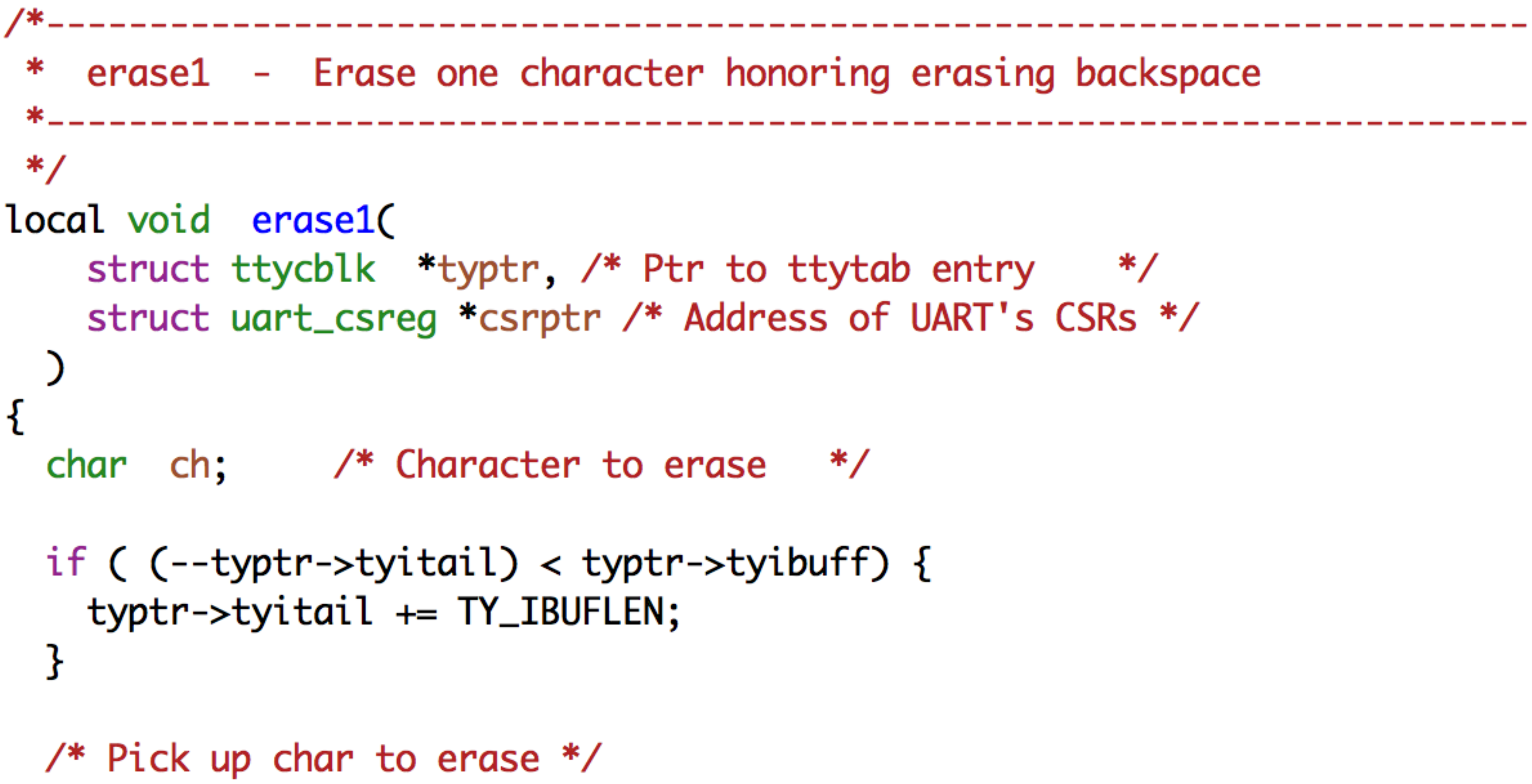
```c
/* EOF character: recognize at beginning of line, but */
/*  print and ignore otherwise.     */

if (ch == typtr->tyeofch && typtr->tyeof) {
  if (typtr->tyiecho) {
    echoch(ch, typtr, csrptr);
  }
  if (typtr->tyicursor != 0) {
    return;
  }
  *typtr->tyitail++ = ch;
  signal(typtr->tyisem);
  return;
}
```

ttyhandle_in.c



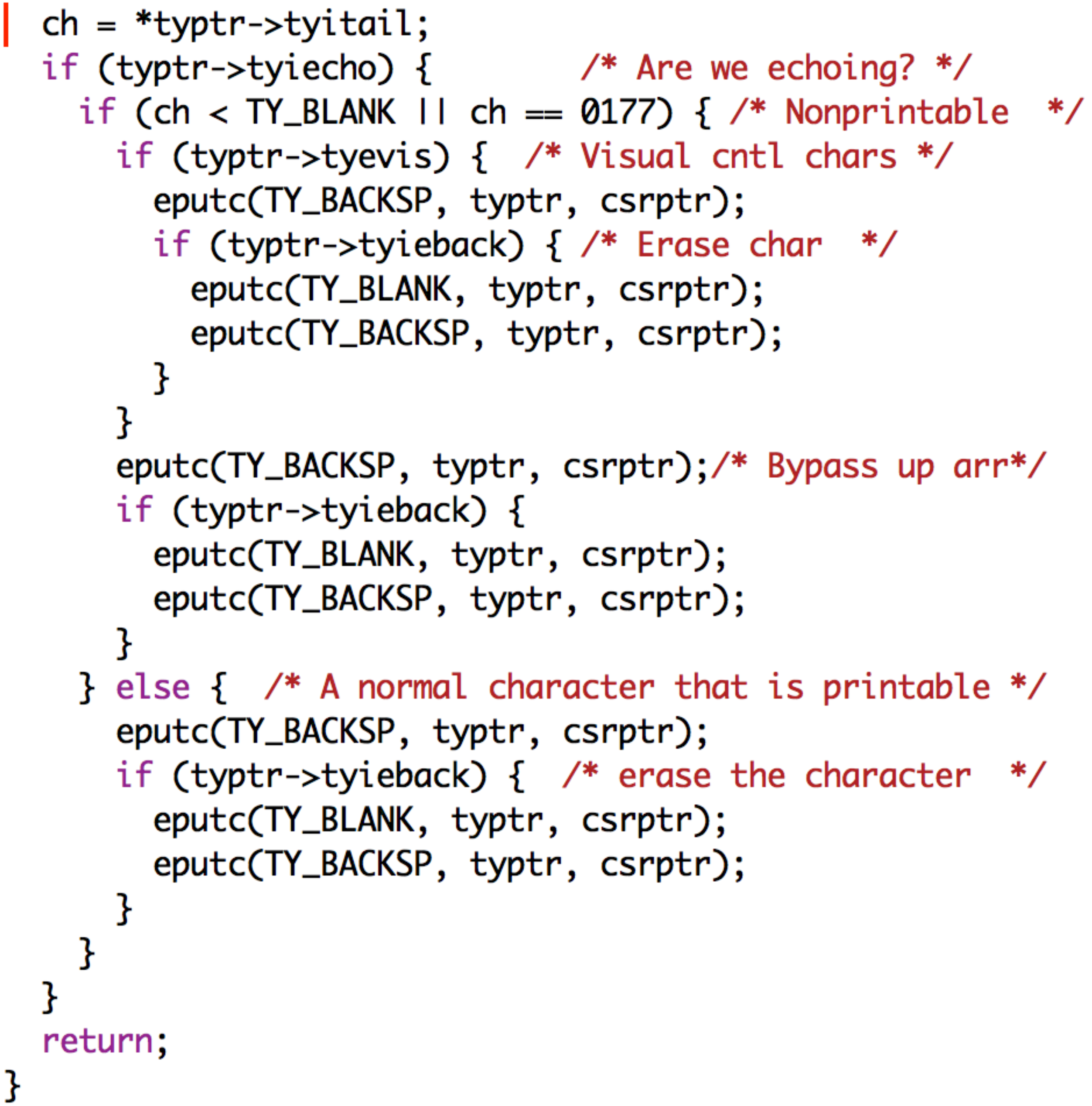
```
    /* Echo the character */

    if (typtr->tyiecho) {
      echoch(ch, typtr, csrptr);
    }

    /* Insert in the input buffer */

    typtr->tyicursor++;
    *typtr->tyitail++ = ch;

    /* Wrap around if needed */

    if (typtr->tyitail >= &typtr->tyibuff[TY_IBUFLEN]) {
      typtr->tyitail = typtr->tyibuff;
    }
    return;
  }
}
```

ttyhandle_in.c (erase1)



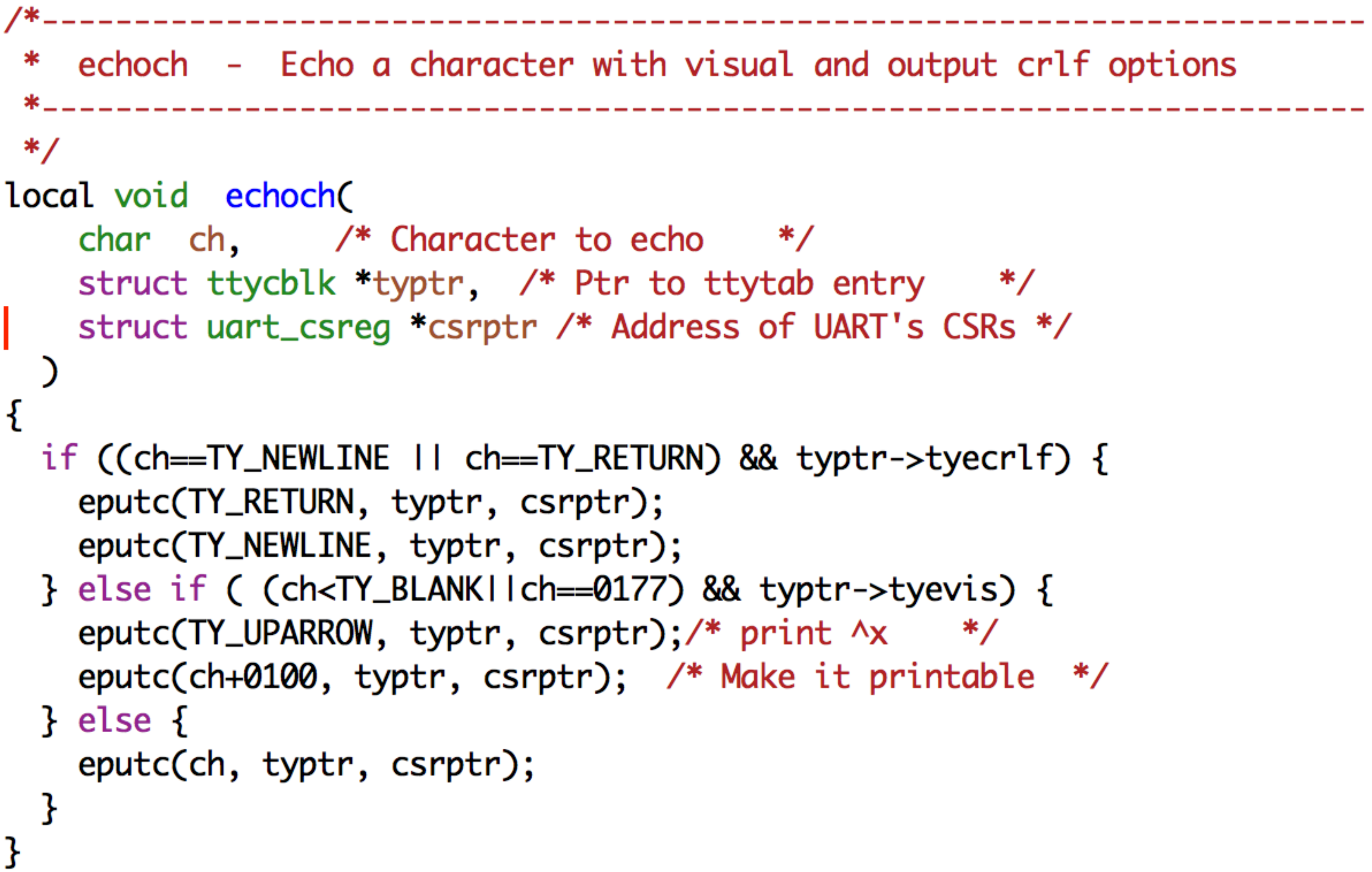
```
/*------------------------------------------------------------------------
 *  erase1  -  Erase one character honoring erasing backspace
 *------------------------------------------------------------------------
 */
local void  erase1(
    struct ttycblk  *typtr, /* Ptr to ttytab entry    */
    struct uart_csreg *csrptr /* Address of UART's CSRs */
  )
{
  char  ch;      /* Character to erase   */

  if ( (--typtr->tyitail) < typtr->tyibuff) {
    typtr->tyitail += TY_IBUFLEN;
  }

  /* Pick up char to erase */
```

ttyhandle_in.c (erase1)



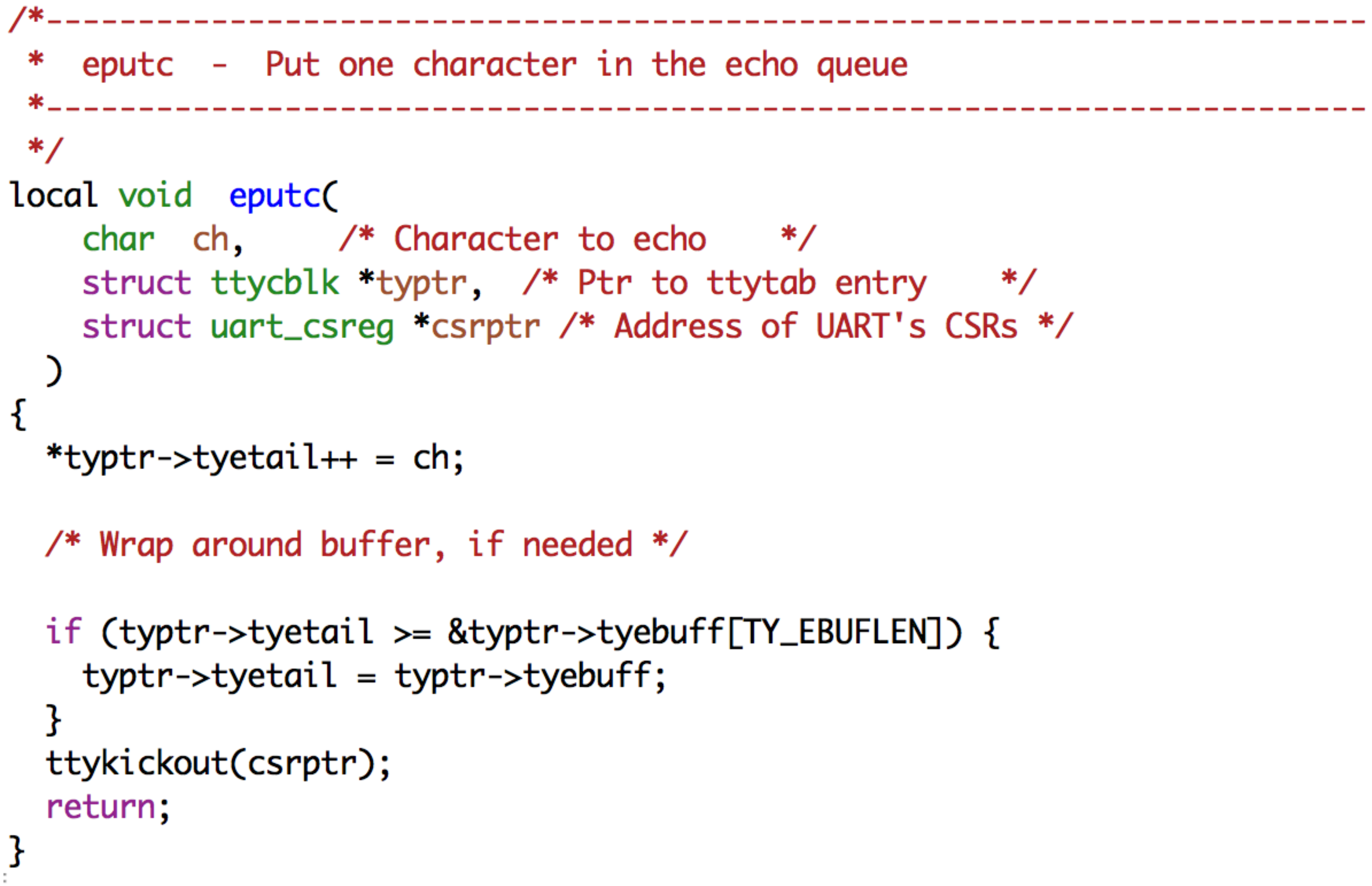
```
  ch = *typtr->tyitail;
  if (typtr->tyiecho) {        /* Are we echoing? */
    if (ch < TY_BLANK || ch == 0177) { /* Nonprintable  */
      if (typtr->tyevis) {  /* Visual cntl chars */
        eputc(TY_BACKSP, typtr, csrptr);
        if (typtr->tyieback) { /* Erase char  */
          eputc(TY_BLANK, typtr, csrptr);
          eputc(TY_BACKSP, typtr, csrptr);
        }
      }
      eputc(TY_BACKSP, typtr, csrptr);/* Bypass up arr*/
      if (typtr->tyieback) {
        eputc(TY_BLANK, typtr, csrptr);
        eputc(TY_BACKSP, typtr, csrptr);
      }
    } else {  /* A normal character that is printable */
      eputc(TY_BACKSP, typtr, csrptr);
      if (typtr->tyieback) {  /* erase the character  */
        eputc(TY_BLANK, typtr, csrptr);
        eputc(TY_BACKSP, typtr, csrptr);
      }
    }
  }
  return;
}
```

ttyhandle_in.c (echoch)



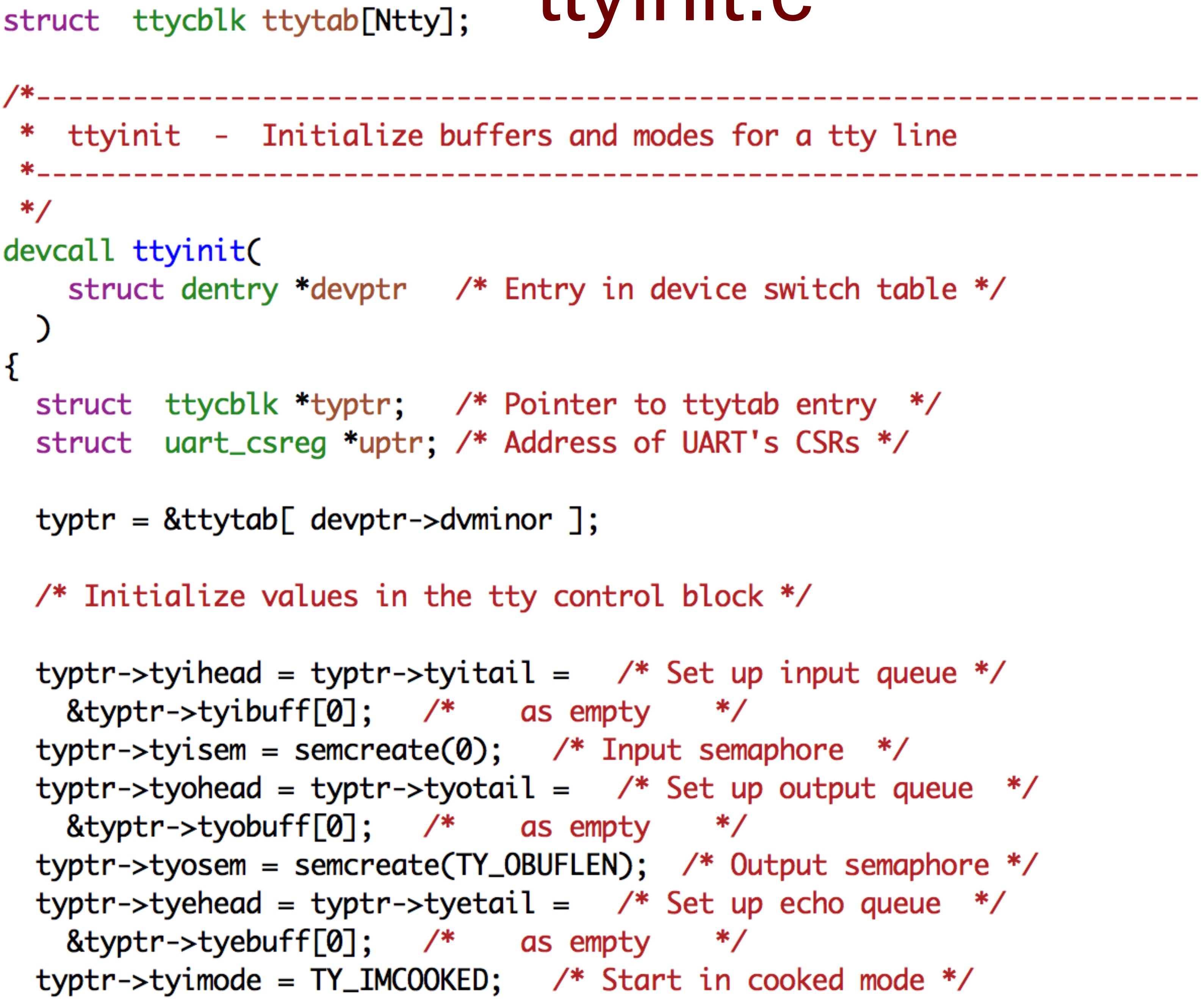
```
/*------------------------------------------------------------------------
 *  echoch  -  Echo a character with visual and output crlf options
 *------------------------------------------------------------------------
 */
local void  echoch(
    char  ch,     /* Character to echo    */
    struct ttycblk *typtr,  /* Ptr to ttytab entry    */
    struct uart_csreg *csrptr /* Address of UART's CSRs */
  )
{
  if ((ch==TY_NEWLINE || ch==TY_RETURN) && typtr->tyecrlf) {
    eputc(TY_RETURN, typtr, csrptr);
    eputc(TY_NEWLINE, typtr, csrptr);
  } else if ( (ch<TY_BLANK||ch==0177) && typtr->tyevis) {
    eputc(TY_UPARROW, typtr, csrptr);/* print ^x    */
    eputc(ch+0100, typtr, csrptr);  /* Make it printable  */
  } else {
    eputc(ch, typtr, csrptr);
  }
}
```

ttyhandle_in.c (eputc)



```
/*------------------------------------------------------------------------
 *  eputc  -  Put one character in the echo queue
 *------------------------------------------------------------------------
 */
local void  eputc(
    char  ch,     /* Character to echo    */
    struct ttycblk *typtr,  /* Ptr to ttytab entry    */
    struct uart_csreg *csrptr /* Address of UART's CSRs */
  )
{
  *typtr->tyetail++ = ch;

  /* Wrap around buffer, if needed */

  if (typtr->tyetail >= &typtr->tyebuff[TY_EBUFLEN]) {
    typtr->tyetail = typtr->tyebuff;
  }
  ttykickout(csrptr);
  return;
}
```

ttyinit.c

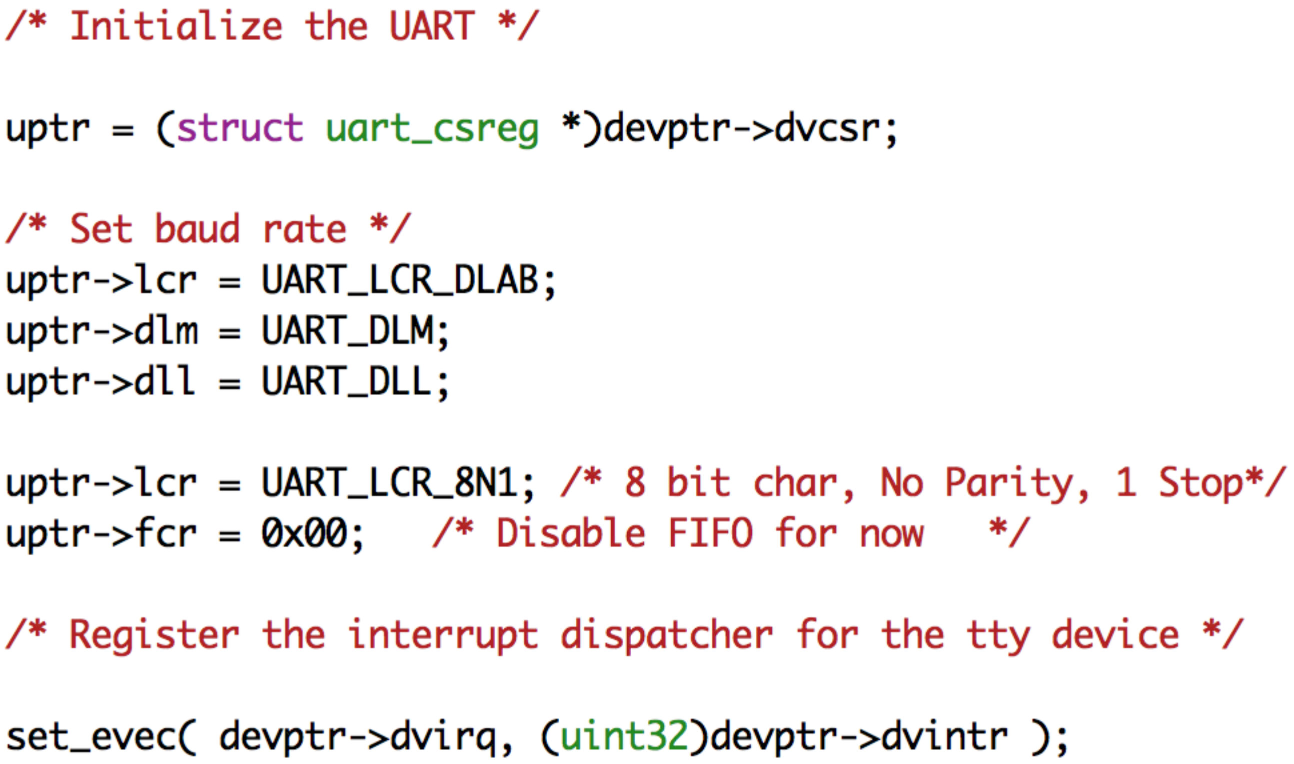


```
struct  ttycblk ttytab[Ntty];

/*------------------------------------------------------------------------
 *  ttyinit  -  Initialize buffers and modes for a tty line
 *------------------------------------------------------------------------
 */
devcall ttyinit(
    struct dentry *devptr   /* Entry in device switch table */
  )
{
  struct  ttycblk *typtr;   /* Pointer to ttytab entry  */
  struct  uart_csreg *uptr; /* Address of UART's CSRs */

  typtr = &ttytab[ devptr->dvminor ];

  /* Initialize values in the tty control block */

  typtr->tyihead = typtr->tyitail =   /* Set up input queue */
    &typtr->tyibuff[0];    /*     as empty     */
  typtr->tyisem = semcreate(0);    /* Input semaphore  */
  typtr->tyohead = typtr->tyotail =   /* Set up output queue  */
    &typtr->tyobuff[0];    /*     as empty     */
  typtr->tyosem = semcreate(TY_OBUFLEN);  /* Output semaphore */
  typtr->tyehead = typtr->tyetail =   /* Set up echo queue  */
    &typtr->tyebuff[0];    /*     as empty     */
  typtr->tyimode = TY_IMCOOKED;    /* Start in cooked mode */
```

ttyinit.c

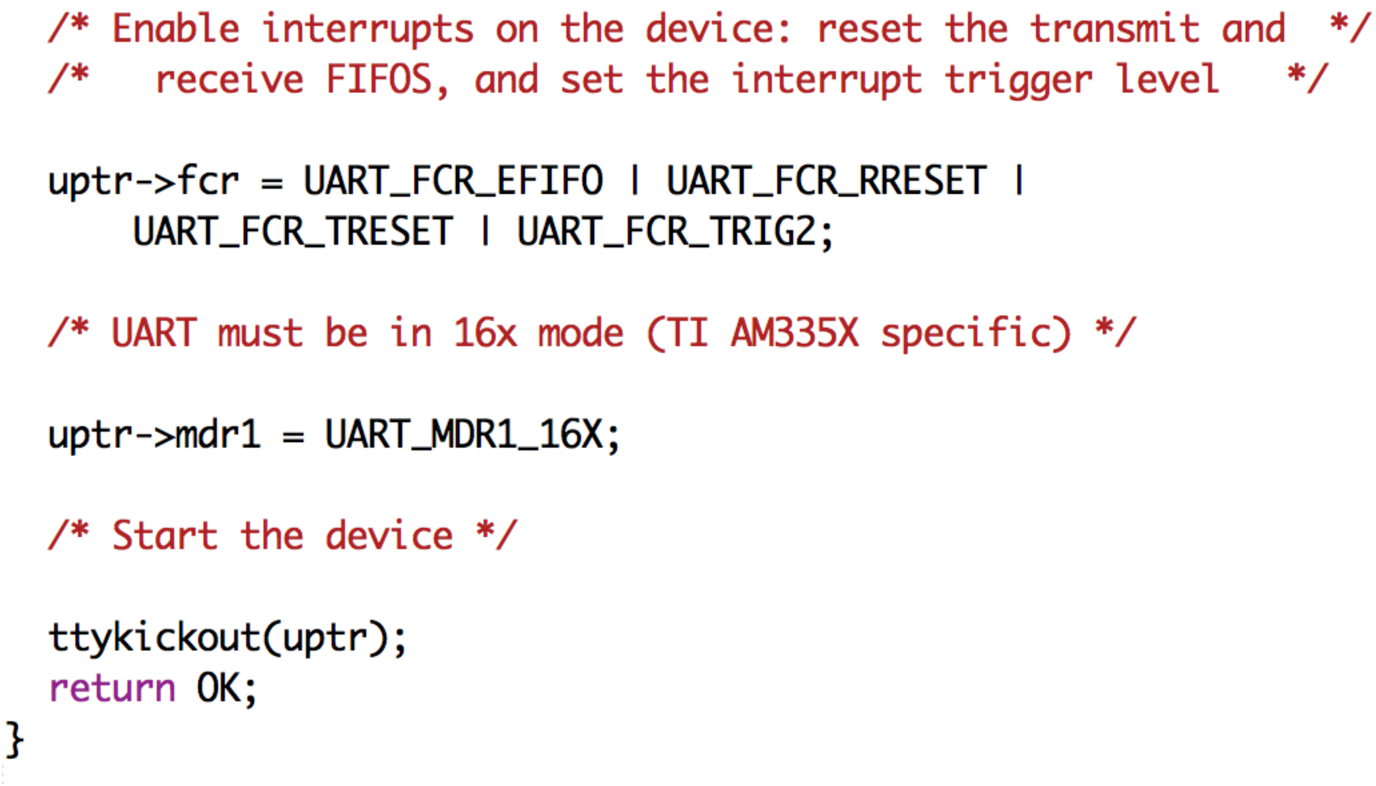
```
typtr->tyimode = TY_IMCOOKED;   /* Start in cooked mode */
typtr->tyiecho = TRUE;      /* Echo console input */
typtr->tyieback = TRUE;     /* Honor erasing bksp */
typtr->tyevis = TRUE;     /* Visual control chars */
typtr->tyecrlf = TRUE;      /* Echo CRLF for NEWLINE*/
typtr->tyicrlf = TRUE;      /* Map CR to NEWLINE  */
typtr->tyierase = TRUE;     /* Do erasing backspace */
typtr->tyierasec = TY_BACKSP;   /* Erase char is ^H */
typtr->tyeof = TRUE;      /* Honor eof on input */
typtr->tyeofch = TY_EOFCH;    /* End-of-file character*/
typtr->tyikill = TRUE;      /* Allow line kill  */
typtr->tyikillc = TY_KILLCH;    /* Set line kill to ^U  */
typtr->tyicursor = 0;     /* Start of input line  */
typtr->tyoflow = TRUE;      /* Handle flow control  */
typtr->tyoheld = FALSE;     /* Output not held  */
typtr->tyostop = TY_STOPCH;   /* Stop char is ^S  */
typtr->tyostart = TY_STRTCH;    /* Start char is ^Q */
typtr->tyocrlf = TRUE;      /* Send CRLF for NEWLINE*/
typtr->tyifullc = TY_FULLCH;    /* Send ^G when buffer  */
          /*   is full    */
```

ttyinit.c



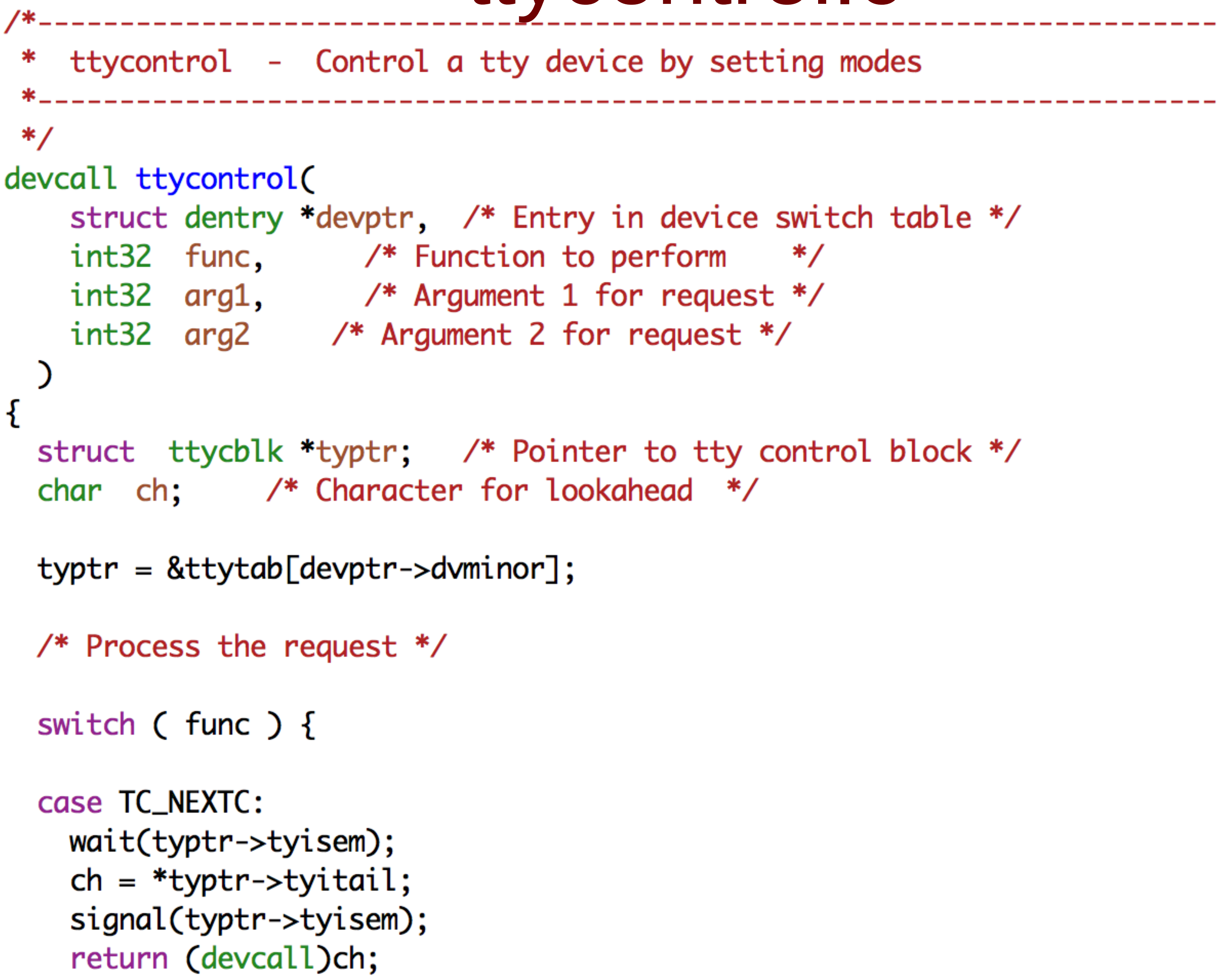
```
/* Initialize the UART */

uptr = (struct uart_csreg *)devptr->dvcsr;

/* Set baud rate */
uptr->lcr = UART_LCR_DLAB;
uptr->dlm = UART_DLM;
uptr->dll = UART_DLL;

uptr->lcr = UART_LCR_8N1; /* 8 bit char, No Parity, 1 Stop*/
uptr->fcr = 0x00;   /* Disable FIFO for now   */

/* Register the interrupt dispatcher for the tty device */

set_evec( devptr->dvirq, (uint32)devptr->dvintr );
```

ttyinit.c



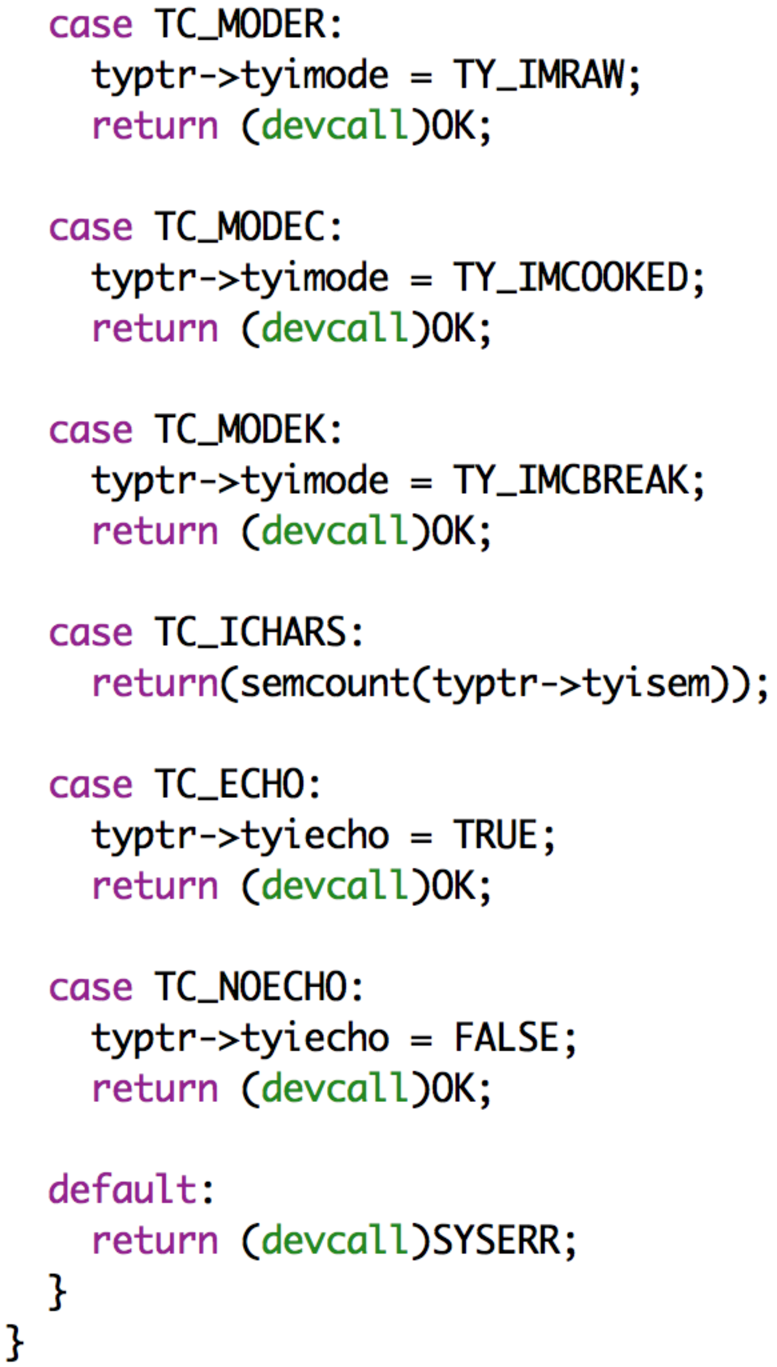
```
  /* Enable interrupts on the device: reset the transmit and  */
  /*   receive FIFOS, and set the interrupt trigger level   */

  uptr->fcr = UART_FCR_EFIFO | UART_FCR_RRESET |
      UART_FCR_TRESET | UART_FCR_TRIG2;

  /* UART must be in 16x mode (TI AM335X specific) */

  uptr->mdr1 = UART_MDR1_16X;

  /* Start the device */

  ttykickout(uptr);
  return OK;
}
```

ttycontrol.c



```
/*------------------------------------------------------------------------
 *  ttycontrol  -  Control a tty device by setting modes
 *------------------------------------------------------------------------
 */
devcall ttycontrol(
    struct dentry *devptr,  /* Entry in device switch table */
    int32  func,      /* Function to perform    */
    int32  arg1,      /* Argument 1 for request */
    int32  arg2     /* Argument 2 for request */
  )
{
  struct  ttycblk *typtr;   /* Pointer to tty control block */
  char  ch;     /* Character for lookahead  */

  typtr = &ttytab[devptr->dvminor];

  /* Process the request */

  switch ( func ) {

  case TC_NEXTC:
    wait(typtr->tyisem);
    ch = *typtr->tyitail;
    signal(typtr->tyisem);
    return (devcall)ch;
```

ttycontrol.c



```
  case TC_MODER:
    typtr->tyimode = TY_IMRAW;
    return (devcall)OK;

  case TC_MODEC:
    typtr->tyimode = TY_IMCOOKED;
    return (devcall)OK;

  case TC_MODEK:
    typtr->tyimode = TY_IMCBREAK;
    return (devcall)OK;

  case TC_ICHARS:
    return(semcount(typtr->tyisem));

  case TC_ECHO:
    typtr->tyiecho = TRUE;
    return (devcall)OK;

  case TC_NOECHO:
    typtr->tyiecho = FALSE;
    return (devcall)OK;

  default:
    return (devcall)SYSERR;
  }
}
```

Summary

- Much complexity for a very simple, static device
- Device drivers are complex
- Upper and lower halves of a device driver work together in decoupled fashion
- The magic of typing into a terminal with echoing and backspacing is not trivial!