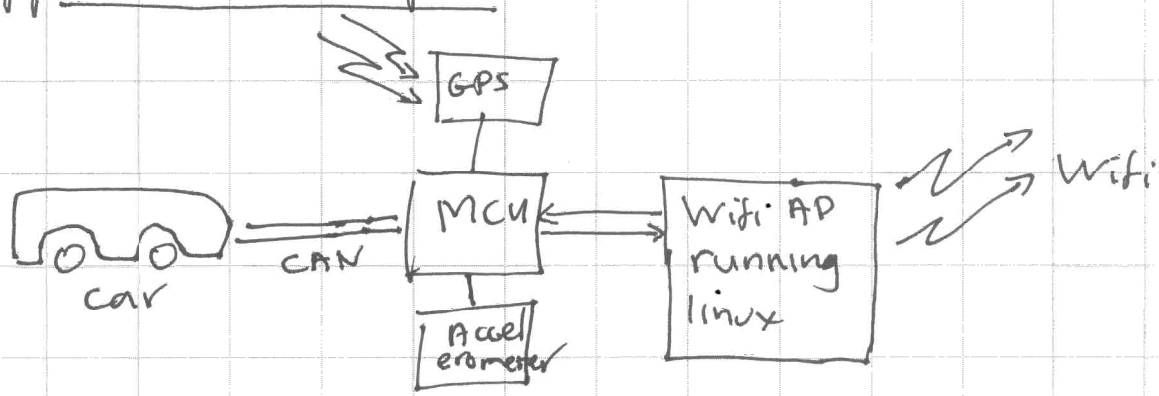
Embedded [illegible]: Asynchronous Serial Comm.

The goal: to get data out of the MCU into a PC.

Useful (very!) for debugging

Some applications need this (and PC→MCU or MCU→MCU, but better ways for this)

Application Example:



The peripheral is called a UART

(universal asynch. receiver transmitter)

Asynchronous serial data transmission

- ~~bit rate~~ ~~Baud rate~~ ~~must be known~~ ~~to both sides.~~

PC operating systems are usually limited to standard speeds: 4800 b/s, 9600, 19200, 38400, 57600, 115200

MCUs can support many speeds

- word size can range from 5 to 9 bits, 8 is standard today

- start bit is always present. When there is no traffic, signal is high ("mark"); transmission begins with the high → low transition at beginning of start bit; can be at any time (idle)

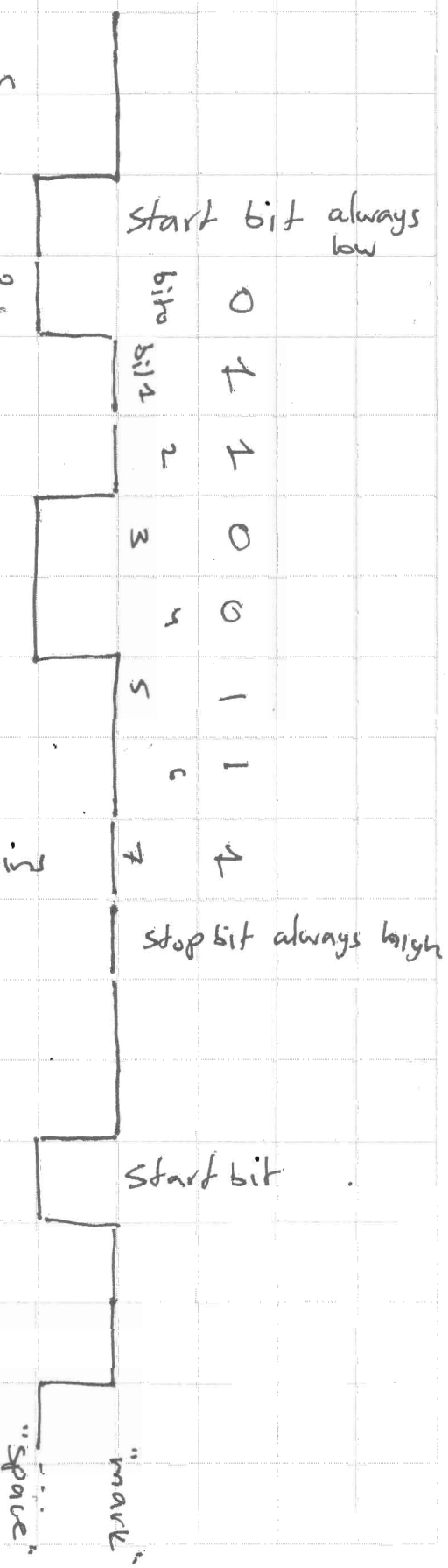
- "stop bit" is the minimum high period after a word; can be 1, 1.5, or ~~2~~ 2 bit periods can be ~~was~~ longer.

- optional parity bit before stop, even, odd, or none (most common)

~~8 bits, no parity, 1 stop bit~~

fitted time quanta

$\frac{1}{\text{bitrate}}$ s e.g. $\frac{1}{9600}$ s

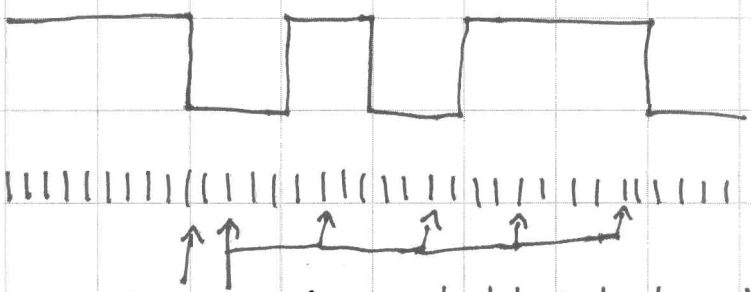


Receiver & transmitter (RX, TX) must agree on: speed (bit period), word size, stop bits, parity

eg 9600 8N1 (8 bits, no parity, 1 stop bit)

Usually <u>full duplex</u>: one signal in each direction, unrelated in terms of timing but same parameters.

How the receiver does?

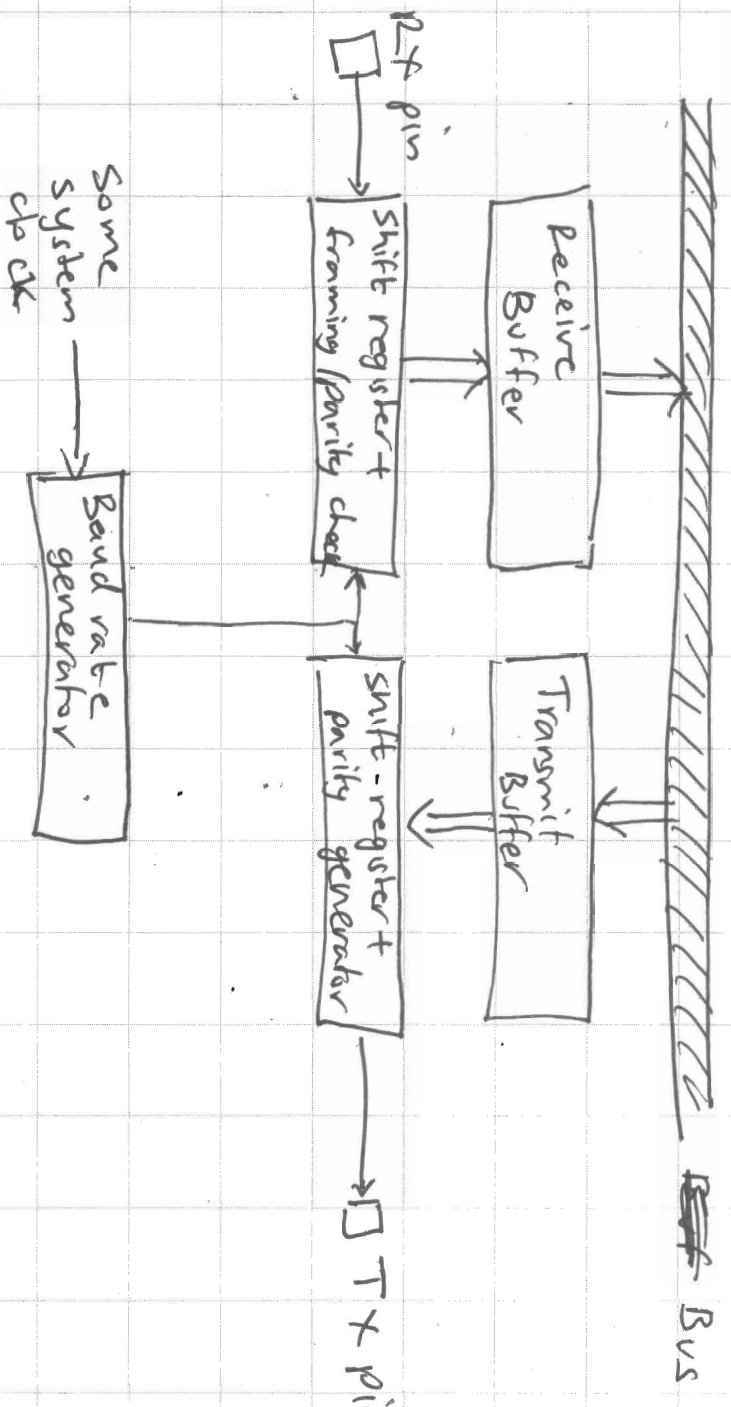


Implications: works well when UART clock is higher than bit rate and an exact multiple of it.

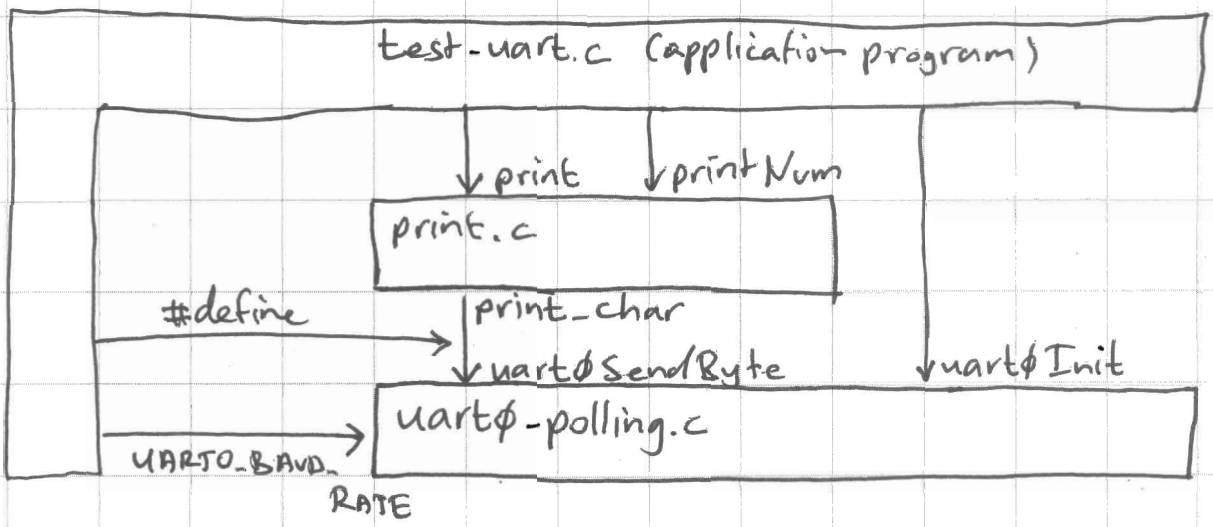
Works otherwise but more errors, especially if both slow & not a multiple

Transmitter: inexact bitrate if clock is not an exact multiple of the bit rate.

Structure of a simple UART



Developing a super-simple UART driver



test_uart.c:

```
#include <stdint.h>
#include <lpc2000/io.h>

volatile int dummy;
const int delay = 100000;

#define CLOCKS_PCLK 3000000

#define UART0_BAUD_RATE 19200
#include "uart0-polling.c"

#define  print_char(x) uart0SendByte(x)
#include "print.c"

int main(void) {
  uart0Init();

  IODIR0 = BIT10;

  while (1) {
    uint32_t i;
    for (i=0; i<delay; i++) dummy=1;
    IOPIN0 |=   BIT10;
    for (i=0; i<delay; i++) dummy=1;
    IOPIN0 &=  ~BIT10;

    print("uart test\n");
  }
}
```

what for?

configure the driver

connect

LED blinking not essential but helpful

Print formatting

```
static uint8_t _printBuffer[12];

static void print(uint8_t* s) {
  for (; *s; s++)
    print_char(*s);
}

static void printHex(uint32_t v, uint8_t digits) {
  int8_t i;
  uint8_t d;

  for (i=0; i<digits; i++) {
    d = v & 0x0000000F;
    _printBuffer[i] = d < 10 ? d + '0' : d - 10 + 'A';
    v >>= 4;
  }

  for (i=digits-1; i>=0; i--)
    print_char(_printBuffer[i]);
}

static void printNum(int32_t v) { ... }
```

Why not use printf?

- We can
- Standard printf is a large function: a lot of code space & a lot of buffer space
- not an issue on large MCU's but too large on small ones
- Some times there are multiple printf libraries with different capabilities/sizes.
- printf often calls putchar() that you need to define

My driver conventions

- Drivers are #include'd into main application program, normally without headers (unless there is a circular dependency)
  - This allows the compiler maximum opportunities to optimize
  - Acceptable compile times for small systems
  - Global variables and all functions ~~define~~ declared static ⇒ perfect dead code/variable elimination
- Exported names start with driver name & are capitalized, e.g. Uart0SendByte
- Private names start with an underscore & module name, e.g. _printBuffer
- Services that the driver needs use underscores to separate words: print_char
  UART0_BAUD_RATE
  - The application program should #define these.

The UART driver itself:

```
static void uart0Init() {
  uint32_t divisor;

  U0LCR = BIT1 | BIT0 /* 8-bit words                                    */
  | BIT7;             /* enable programming of divisor latches  */

  {
    ...
    /* complex calculation to generate U0DLM, U0DLL, U0FDR
       from CLOCKS_PCLK and UART0_BAUD_RATE */
    }
    U0DLM = (min_divisor & 0x0000FF00) >> 8;
    U0DLL = (min_divisor & 0x000000FF);
    U0FDR = min_divaddval | (min_mulval << 4);
  }

  U0LCR &= ~BIT7; /* disable programming of divisor latches */

  U0FCR = BIT0 | BIT1 | BIT2; /* FIFO control: enable & reset     */

  PINSEL0 &= ~(BIT1 | BIT3); /* select UART function                */
  PINSEL0 |= BIT0 | BIT2; /* for pins P0.0 and P0.1                */
}

static void uart0SendByte(uint8_t c) {
  while (!(U0LSR & BIT5)); wait for TX buffer not full
  U0THR = c; put byte in TX buffer
}
```

Baud-rate control

$$\text{Baud Rate} = \frac{\text{CLOCKS\_PCLK}}{16\,(256 \cdot U0DLM + U0DLL)} \times \frac{\text{MULVAL}}{(\text{MULVAL} + \text{DIVADDVAL})}$$

$0 \le U0DLM, U0DLL \le 255$ (8-bit SFR's)

$1 \le \text{MULVAL} \le 15$ (4-bit pieces of U0FDR)

$1 \le \text{DIVADDVAL} \le 15$

Two more rules...

Do this computation at compile time!!!
(der tools often help)

why 8-bit SFR's in a 32-bit MCU? An old UART design

Advanced UART features

- Hardware flow control: Additional signals to prevent overruns & enable other features (half duplex)
  CTS (clear-to-send) & RTR (ready-to-receive)
  input output

  Also RTS, DTR, DSR, DCD, RI

  Sometimes used in embedded ⟷ PC applications to send unrelated signals to/from PC, e.g. reset to the embedded system.

- Deep transmit/receive fifo buffers

- Majority-vote bit detection (requires a faster clock)

- Auto-Baud: automatically set Band-rate generator on reception of next char; must be a specific character, usually 0x55 or 'A'; reduces cost by eliminating need for a crystal

- Multiple UARTs; same SFR layout.

- Break: too many 0's to wake up receiver, start a frame