

Lecture 14: Lab 3 + UART communication

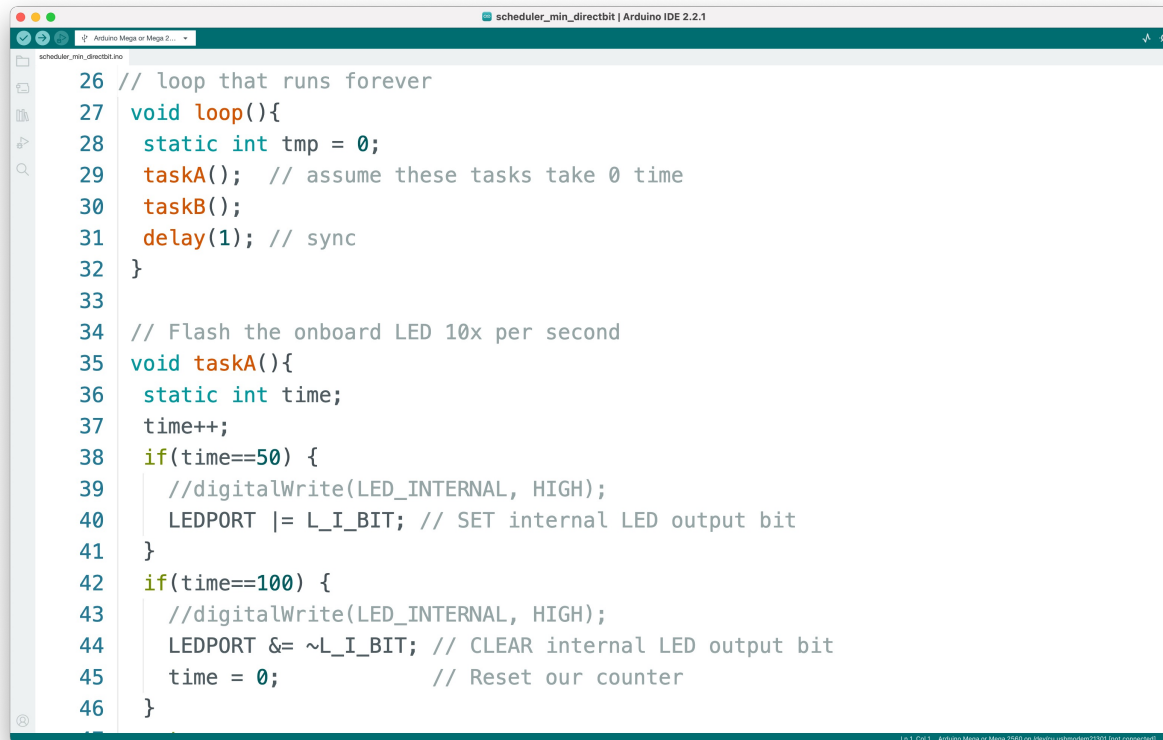
Vikram Iyer

Monday	Tuesday	Wednesday	Thursday	Friday
<p>9:30-11:30 OH (Deeksha) 06 ECE 345</p> <p>11:30-13:30 OH (Alex) ECE 345</p> <p>23:59 Lab 2 due</p>	<p>13:00-15:00 OH (Zach) 07 ECE 345</p>	<p>9:30-11:30 OH (Deeksha) 08 ECE 345</p> <p>12:30-14:20 Lecture MOR 230 <i>Lecture 12: Midterm Review slides, annotated</i></p> <p>14:00-15:00 OH (Vikram) ECE 345</p>	<p>10:30-12:30 OH (Vikram) 09 ECE 345</p> <p>12:30-14:30 OH (Alex) ECE 345</p>	<p>12:30-14:00 Midterm 10 MOR 230 (Lecture room)</p> <p>12:30-15:30 Midterm (DRS) ECE 345 (Lab room)</p>
<p>9:30-11:30 OH (Deeksha) 13 ECE 345</p> <p>11:30-13:30 OH (Alex) ECE 345</p>	<p>14</p>	<p>9:30-11:30 OH (Deeksha) 15 ECE 345</p> <p>12:30-14:20 Lecture Zoom link <i>Lecture 13: I2C and SPI Communication slides</i></p> <p>13:00-15:00 OH (Vikram) Zoom link</p>	<p>10:00-12:00 OH (Deeksha) 16 ECE 345</p> <p>12:30-14:30 OH (Alex) ECE 345</p>	<p>12:30-14:20 Lecture 17 MOR 230 <i>Lecture 14: Lab 3 + UART Communication slides, code</i></p> <p>14:00-15:00 OH (Vikram) ECE 345</p>
<p>9:30-11:30 OH (Deeksha) 20 ECE 345</p> <p>11:30-13:30 OH (Alex) ECE 345</p>	<p>13:00-15:00 OH (Deeksha) 21 ECE 345</p>	<p>9:30-11:30 OH (Deeksha) 22 ECE 345</p> <p>12:30-14:20 Lecture MOR 230 <i>Lecture 15: Intro to FreeRTOS</i></p> <p>14:00-15:00 OH (Vikram) ECE 345</p> <p>23:59 Lab 3 due</p>	<p>10:00-12:00 OH (Zach) 23 ECE 345</p> <p>12:30-14:30 OH (Alex) ECE 345</p>	<p>12:30-14:20 Lecture 24 MOR 230 <i>Lecture 16: Intro to Critical Sections and Semaphores</i></p> <p>14:00-15:00 OH (Vikram) ECE 345</p>

Round Robin Scheduler

```
while(1) {  
    taskA();  
    taskB();  
    taskC();  
    time_delay();  
}
```

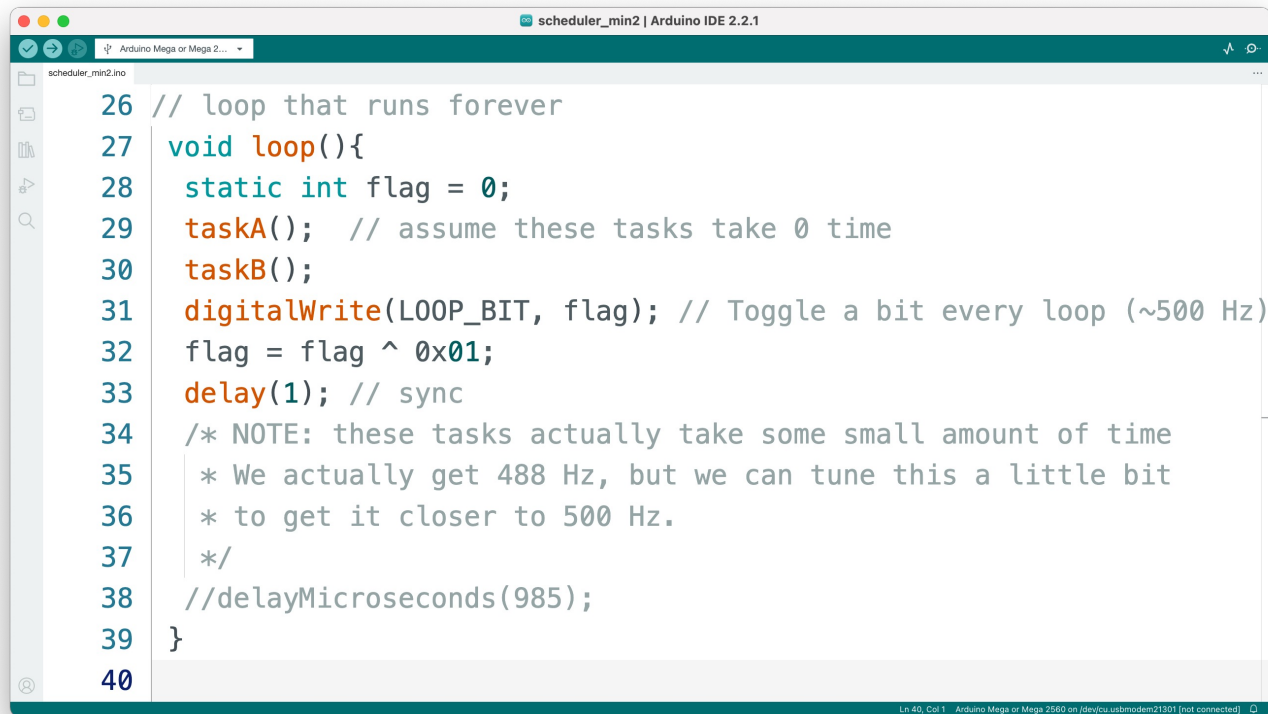
Round Robin Scheduler



```
scheduler_min_directbit | Arduino IDE 2.2.1
scheduler_min_directbit.ino
26 // loop that runs forever
27 void loop(){
28   static int tmp = 0;
29   taskA(); // assume these tasks take 0 time
30   taskB();
31   delay(1); // sync
32 }
33
34 // Flash the onboard LED 10x per second
35 void taskA(){
36   static int time;
37   time++;
38   if(time==50) {
39     //digitalWrite(LED_INTERNAL, HIGH);
40     LEDPORT |= L_I_BIT; // SET internal LED output bit
41   }
42   if(time==100) {
43     //digitalWrite(LED_INTERNAL, HIGH);
44     LEDPORT &= ~L_I_BIT; // CLEAR internal LED output bit
45     time = 0; // Reset our counter
46   }
47 }
```

Ln 1, Col 1 - Arduino Mega or Mega 2560 on /dev/ttyUSB0 [not connected]

How fast does it run?



```
26 // loop that runs forever
27 void loop(){
28     static int flag = 0;
29     taskA(); // assume these tasks take 0 time
30     taskB();
31     digitalWrite(LOOP_BIT, flag); // Toggle a bit every loop (~500 Hz)
32     flag = flag ^ 0x01;
33     delay(1); // sync
34     /* NOTE: these tasks actually take some small amount of time
35     * We actually get 488 Hz, but we can tune this a little bit
36     * to get it closer to 500 Hz.
37     */
38     //delayMicroseconds(985);
39 }
40
```

Ln 40, Col 1 Arduino Mega or Mega 2560 on /dev/cu.usbmodem21301 (not connected)

Using function pointers to start a task

We learned that the syntax:

```
type (*name) (type arg1, type arg2, ...)
```

indicates a function pointer called name which can point to any function having the same prototype. For example

```
int (*fp) (char x, int y, double z)
```

```
void start_function(void (*functionPTR) () ) {  
    functionPTR();  
}
```

Keeping lists of functions

We can create an array of function pointers as follows:

```
#define NUMBER_OF_FUNCTIONS 10
```

```
void (*list_of_functions[NUMBER_OF_FUNCTIONS])(void *p)
```

Example:

```
#define NT_RUNNING 10
```

```
void (*runningTasks[NT_RUNNING])(void *p)
```

... a list of all the tasks which are currently running.

We can use a NULL pointer at the end of the list if there are less than NT RUNNING active tasks (similar to the zero byte at the end of a string).

```
//function prototypes for tasks
void taskA(void *p);
void taskB(void *p);
...

// function prototype for
scheduler void scheduler();

#define NTASKS 4

// lets make the task list global
// an array of function pointers,
// each one has a void*
// parameter.

void (*readytasks[NTASKS])
    (void *p);
```

```
//continued...
main() {
    // initialize array of pointers
    // to tasks

    readytasks[0] = taskA;
    readytasks[1] = taskB;
    ...

    // NULL signals the last task
    readytasks[2]= NULL;

    // now start scheduler
    while(1) {
        scheduler();
        time_delay(); // 1 ms timer
    }
} // end of main
```

Scheduler function (pseudocode)

```
scheduler() {  
    static int task_index  
    if(readytasks[task_index] == NULL && task_index != 0) {  
        task_index=0; ← we're at the end restart with task A  
    }  
    if(readytasks[task_index] == NULL && task_index == 0){  
        // figure out something to do because  
        // there are no tasks to run!  
    } else {  
        start function(readytasks[task_index]);  
        task_index++;  
    }  
    // Round Robin/we're taking turns  
    return;  
}
```

Part III: Manipulating the Task Lists

```
halt_me() {
    // 1. Identify which task is currently running (i.e.
    //    look at task_index)

    // 2. Copy the function pointer from
    //    readytasks[task_index] to the
    //    haltedtasks array

    // 3. Move the remaining tasks up in readytasks[] to
    //    fill the empty hole and copy NULL into the
    //    last element.

    return;
}
```

Sleep function (pseudocode)

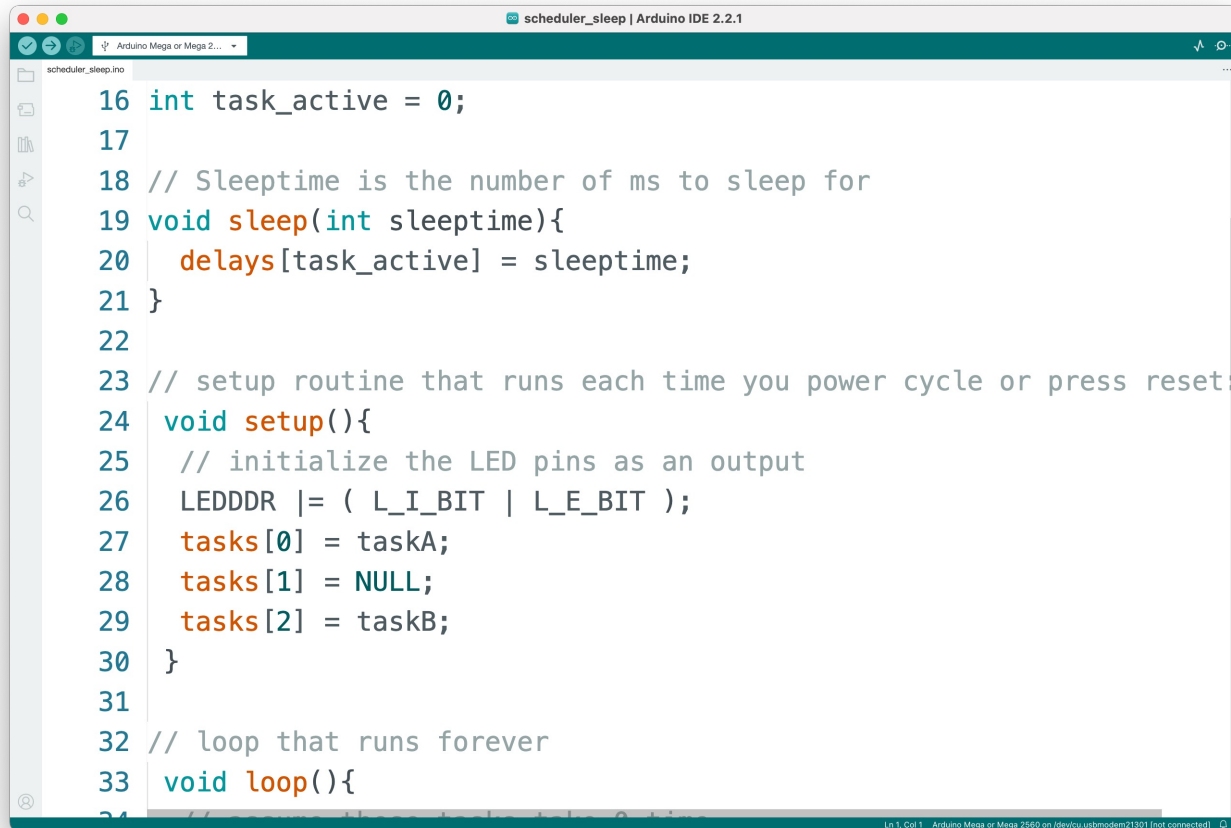
```
sleep(int d) {  
    // 1. Copy function pointer from  
    //    readytasks[task_index] to the  
    //    waitingtasks[] array.  
    // 2. Clean up readytasks[] as in halt_me();  
    //    copy d into the delays array with the same  
    //    index as the function pointer has in  
    //    waitingtasks[]  
}
```

↪ number of cycles to sleep

Keeping track of sleep delays

```
// continued...
// 5. for each element of waitingtasks which is
//     not NULL: decrement the delay value d.
//     if (d==0) move the function pointer to the
//     end of the readytasks[] array and remove
//     it from waitingtasks.
// end of scheduler
return;
}
```

Sleep example



```
scheduler_sleep | Arduino IDE 2.2.1
scheduler_sleep.ino
16 int task_active = 0;
17
18 // Sleep time is the number of ms to sleep for
19 void sleep(int sleeptime){
20     delays[task_active] = sleeptime;
21 }
22
23 // setup routine that runs each time you power cycle or press reset
24 void setup(){
25     // initialize the LED pins as an output
26     LEDDR |= ( L_I_BIT | L_E_BIT );
27     tasks[0] = taskA;
28     tasks[1] = NULL;
29     tasks[2] = taskB;
30 }
31
32 // loop that runs forever
33 void loop(){
34     // ...

```

Alternative Data Structure for Scheduling

Let's set up a struct **Task Control Block (TCB)** which contains everything we need to track about a Task:

```
typedef struct TCBstruct {
    void (*ftpr)(void *p);    // the function pointer
    void *arg_ptr;           // the argument pointer
    unsigned short int state; // the task state
    unsigned int delay;      // the task state
} ;
```

```
#define STATE_RUNNING 0
#define STATE_READY 1
#define STATE_WAITING 2 ← sleep
#define STATE_INACTIVE 3 ← halt
TCBStruct TaskList[N_MAX_TASKS];

// Then we can set up the task // list as follows:
int j=0;
int task_B_Arg;

TaskList[j].ftpr = task_A();
TaskList[j].arg_ptr = NULL;
TaskList[j].state = STATE_INACTIVE;
TaskList[j].delay = 0;
j++;
```

```
// continued... start task B
TaskList[j].fptr = task_B();

// (example: let's say we need an arg value of 56)
task_B_Arg = 56;
int *ip = &task_B_Arg;
TaskList[j].arg_ptr = (void*)ip;
TaskList[j].state = STATE_READY;
TaskList[j].delay = 0;
j++;

TaskList[j].fptr = NULL;           // marks end of list

... maybe other tasks    ...
```

Examples of `halt_me()`, `start_task()` and `delay()`

```
halt_me() {
    TaskList[t_curr].state = STATE_INACTIVE;
}

start_task(int task_id) {
    TaskList[task_id].state = STATE_READY;
}

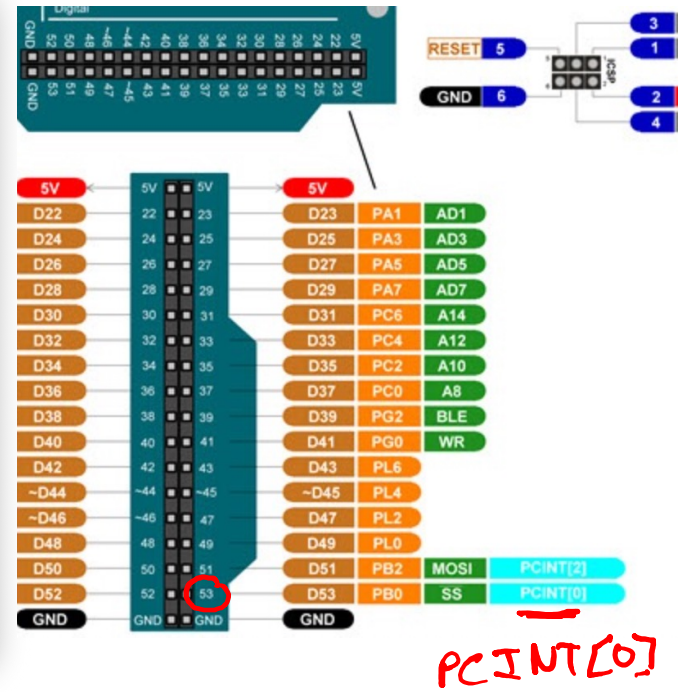
sleep(int d) {
    TaskList[t_curr].delay = d;
    TaskList[t_curr].state = STATE_WAITING;
}
```

Interrupt example (Pin change interrupt on pin 53)

```

1 #define LED_INTERNAL 7 // Pin 13 = bit 7 MEGA, bit 5 Uno
2 #define LED_EXTERNAL 4 // Pin 10 = bit 4 MEGA, bit 2 Uno
3 #define LEDPORT PORTB
4 #define LEDDDR DDRB
5 #define L_I_BIT 1<<LED_INTERNAL
6 #define L_E_BIT 1<<LED_EXTERNAL
7
8 ISR (PCINT0_vect) {
9 // handle pin change interrupt for D8 to D13 here
10 LEDPORT ^= L_I_BIT; // SET internal LED output bit
11 } // end of PCINT0_vect
12
13 void setup () {
14 // initialize the LED pins as an output
15 LEDDDR |= ( L_I_BIT | L_E_BIT );
16

```

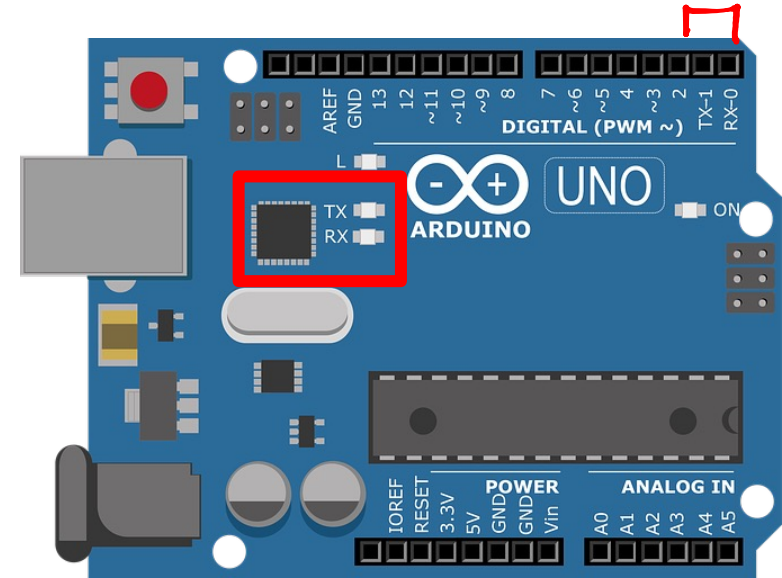


Serial Communication: RS232C

Goal: Send bits from point A to point B one at a time

RS232C: An international standard for serial communication (1960s). Very old but still influential in today's systems.

- This is how your Arduino communicates with the computer and why we do `Serial.println`
- FTDI chip converts between Serial and USB
- Some microcontrollers have native USB support
- Notation: RX = receive, TX = Transmit



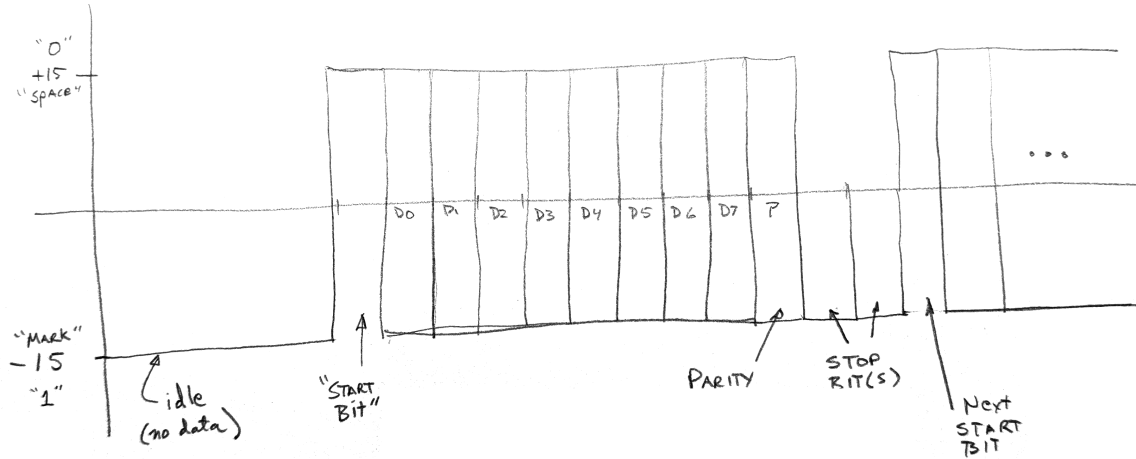
Electrical Signaling

+2.5V — +15V	logic 0	“space”
-2.5V — -15V	logic 1	“mark”

- When first designed the high voltage margin gave more immunity to noise
- These days it's a hassle since many systems don't otherwise need +15V, -15V supplies, modern processes can't support high voltages
- Requires special driver chips for +- voltages.
- Arduino and many other systems use
 - +5V → logic 0
 - 0V → logic 1
- Lower voltage saves power supply costs

Serial Signaling

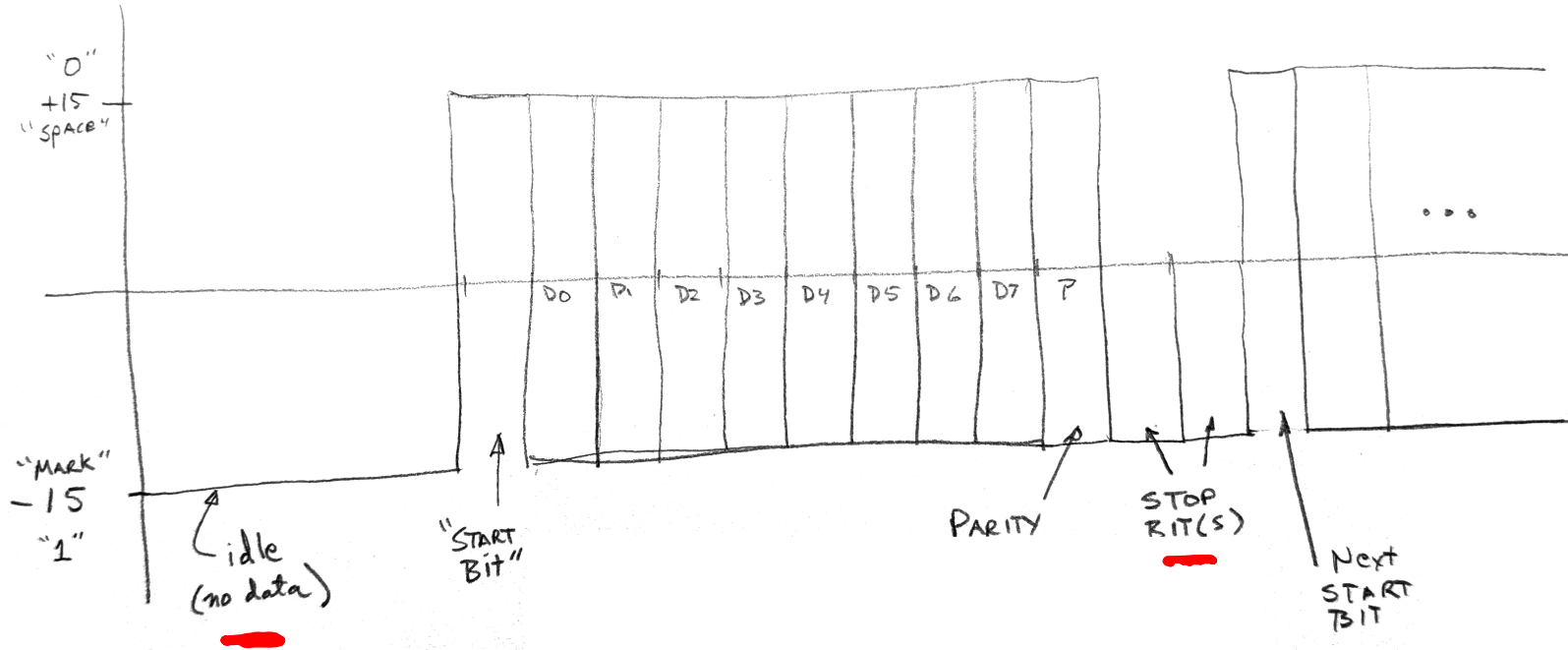
Pos.	Value	Function
0	space	Start Bit
1	X	Data 0 (lsb)
2	X	Data 1
...	X	...
8	X	Data 7 (msb)
9	X	Parity*
10	mark	Stop Bit 1
11	mark	Stop Bit 2*



Idle line in "mark" state (logic 1, -15V)

* Note: Parity and Stop Bit 2 are optional, can have 1.5 Stop Bits.

Serial Signaling



Parity

All communication systems can suffer from errors

Parity bit provides a simple error checking mechanism

Compute by taking the XOR of all data bits:

$$P = D_0 \oplus D_1 \oplus D_2 \dots \oplus D_7$$

Interpretation of XOR(bits) is

"1" if number of ones is ODD

"0" if number of ones is EVEN

Parity Example

D0	D1	D2	XOR(D0,D1,D2)
0	0	0	0
0	<u>0</u>	<u>1</u>	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Speed and Timing

Baud is short for Baudot, 19th century French telegrapher who invented first fixed length alphanumeric code.

Baud or **Baud rate** means number of signal changes on the line per second. Commonly equal to bits per second

Example: Arduino serial port default: 9600 Baud

```
Serial.begin(9600);           //Baud rate = 9600  
Serial.begin(115200);       //Baud rate = 115200
```

Synchronization

There is no common clock between sender and receiver. How does receiver know when to test levels?

A1: Synchronous mode — Continuous stream of characters.

A2: Asynchronous mode — start bit is known to be logic 0. Falling (rising?) edge triggers a local clock in the receiver.

Multiplexing

Sending information in both directions. (Stringing 2 telegraph wires was very expensive!)

Terms:

Simplex One sender and one receiver.

Half Duplex One side can send at a time.

Full Duplex Both sides can send at same time.

Connections

Must have:
TX → RX
RX ← TX



Tx = transmit
Rx = receive
computer

Cables are easier to make if pins are “straight through”. i.e. pin n connects

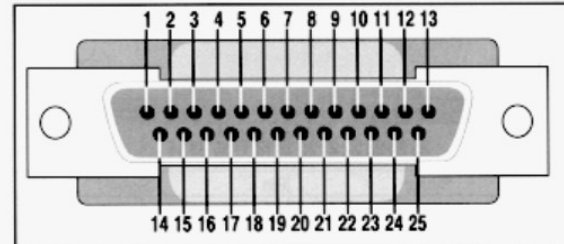
23 pin (ancient, un-needed) 9 pin (IBM PC standard)

Two connector wiring types

DTE: pin 2 = RX, pin 3 = TX “Data Terminal Equipment”

DCE: pin 2 = TX, pin 3 = RX “Data Communications Equipment”

Thus, DCE can talk to DTE with a “straight through” cable.



Pin	Description	EIA CKT	From DCE	To DCE
1	Frame Ground	AA		
2	Transmitted Data	BA		D (Data)
3	Received Data	BB	D	
4	Request to Send	CA		C (Control)
5	Clear to Send	CB	C	
6	Data Set Ready	CC	C	
7	Signal Gnd/Common Return	AB		
8	Rcvd. Line Signal Detector	CF	C	
11	Undefined			
12	Secondary Rcvd. Line Sig. Detector	SCF	C	
13	Secondary Clear to Send	SCB	C	
14	Secondary Transmitted Data	SBA		D
15	Transmitter Sig. Element Timing	DB	T (Timing)	
16	Secondary Received Data	SBB	D	
17	Receiver Sig. Element Timing	DD	T	
18	Undefined			
19	Secondary Request to Send	SCA		C
20	Data Terminal Ready	CD		C
21	Sig. Quality Detector	CG		C
22	Ring Indicator	CE	C	
23	Data Sig. Rate Selector (DCE)	CI	C	
23	Data Sig. Rate Selector (DTE)	CH		C
24	Transmitter Sig. Element Timing	DA		T
25	Undefined			

Fig.6. RS232 DB25 connector

Flow Control

Sometimes a slow device cannot handle data as fast as sender (even at same baud rate).
Receiver needs a way to stop and start sender.

Hardware solution: Additional wires

RTS “Request to Send”. DTE sets this to mark to allow data to be sent to it by DCE.

CTS “Clear to Send”. DCE sets this to mark to allow data to be sent to it by DTE.

Hardware flow control can be enabled/disabled.

Software solution: Special Control Characters

XON (ctl Q) Start Transmitting

XOFF(ctl S) Stop Transmitting

Error Control

Unfortunately, serial communication is subject to errors. Sources:

- Electrical interference
- Long wire lengths
- Ground potential differences between communicating systems.
- Timing errors between sender and receiver clocks (should be within 1%).

Solutions

Character Parity 9th data bit. Very weak and bandwidth hog.

Checksum Add up a group of bytes. Send sum at end of packet.

Compare.

Parity Send parity of some group of bits (rows or columns) in the packet.

CRC Use theory of binary polynomials. Send a code so that combined message is divisible by some known polynomial.

Packets

Grouping bytes into packets gives structure to the serial communication. Packet structure must be same between software both sides.

Example Packet:



Start A character which the software can recognize as the start of a packet. Ideally should not appear anywhere else in packet.

Type Kind of packet.

Length value which tells how long the packet will be.
... some number of data bytes (the payload).

Checksum Checksum computed on previous packet.

Packet Error control

What if a packet is received and the Checksum is wrong?

Normal:

Sender sends the packet and saves the packet.

Receiver checks Checksum and acknowledges receipt of packet.

Sender discards packet and sends next packet.

Error:

Sender sends and saves the packet.

Receiver detects error and sends negative acknowledge.

goto step 1.

ACK/NAK Special short packets

ACK Acknowledge NAK Negative Acknowledge

Start	ACK/NAK	Checksum
-------	---------	----------

Protocol Issues

What Happens If:

Transmission is so unreliable that Checksum fails every time?

ACK/NAK packets are subject to errors?

Do we have to send ACK/NAK on receipt of an ACK/NAK?

Answers to all these questions must be worked out in a protocol spec.