Computer Networks
Data Link Protocol

Ana Aguiar, Eduardo Almeida
DEEC, FEUP

Fall/Winter Semester 2021-22

Thanks to Prof. José Ruela and Prof. Manuel Ricardo for originally developing this assignment

Overview

- Goals
  - Implement a data communication protocol
  - Test protocol with data transfer application

- Competences
  - Understand principles of data communication, layering, interfaces, functionality separation, state machines
  - Compiling and debugging a distributed application in Linux environment
  - Developing a distributed application in a team environment
  - Basic Linux commands, gdb, make

Organisation

- Development Environment
  - PCs with Linux
  - Programming language: C
  - Serial ports: RS-232 (asynchronous communication)

- Groups
  - Groups of 2 students
  - One develops the transmitter
  - Another develops the receiver

Evaluation

- Implementation of the data link protocol
  - Frame synchronisation, error robustness, error correction, ...
- Implementation of the application protocol
  - Control packets, packet numbering
- Testing with ready made application protocol
- Code organisation
  - Modularity and layering, Data link protocol API
- Demonstration

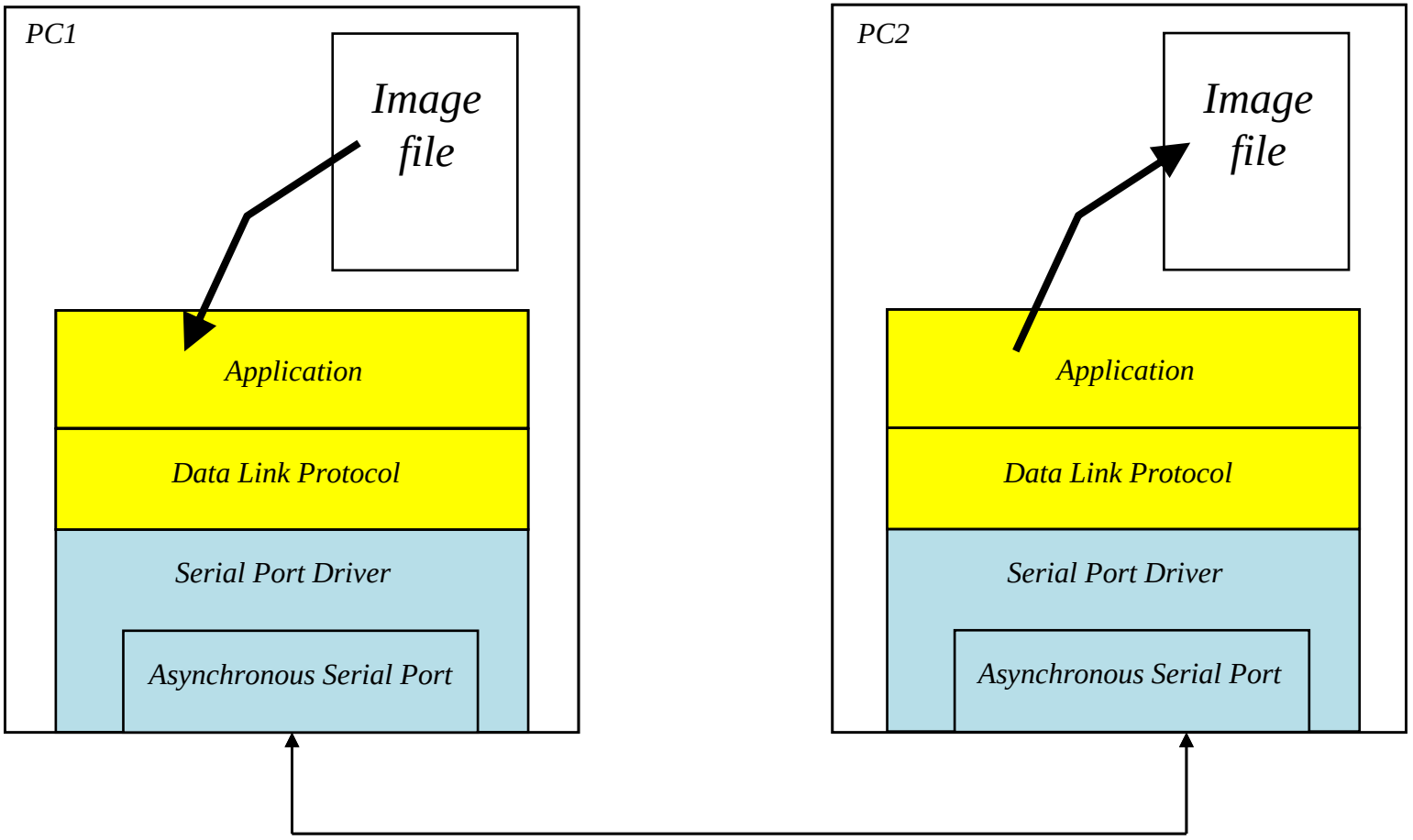
Valorisation Elements

- User parameter selection
  - Baud rate, maximum size of I frames data field size (without stuffing), maximum number of retransmissions (default = 3), time-out interval
- Random error generation in data frames
  - Suggestion: for each correctly received I frame, simulate at the receiver the occurence of errors in header and data field according to pre-defined and independent probabilities, and proceed as if they were real errors
- REJ implementation
- Application verifies data integrity
  - Received file size (real vs. indicated in control packets)
  - Lost or duplicated packets (using packet numbering field)
  - Error recovery – for example, close the connection (DISC), re-establish it (SET) and restart the process
  - Event log (errors
  - Number of retransmitted/ received I frames, number of time-outs, number of sent/ received REJ

Valorisation Elements

- File transmission statistics
  - Error recovery – for example, close the connection (DISC), re-establish it (SET) and restart the process
  - Event log (errors)
  - Number of retransmitted/received I frames, number of time-outs, number of sent/received REJ
- Quantitative experimental results
  - Quantifying the protocol performance for different protocol configurations and channel error rates

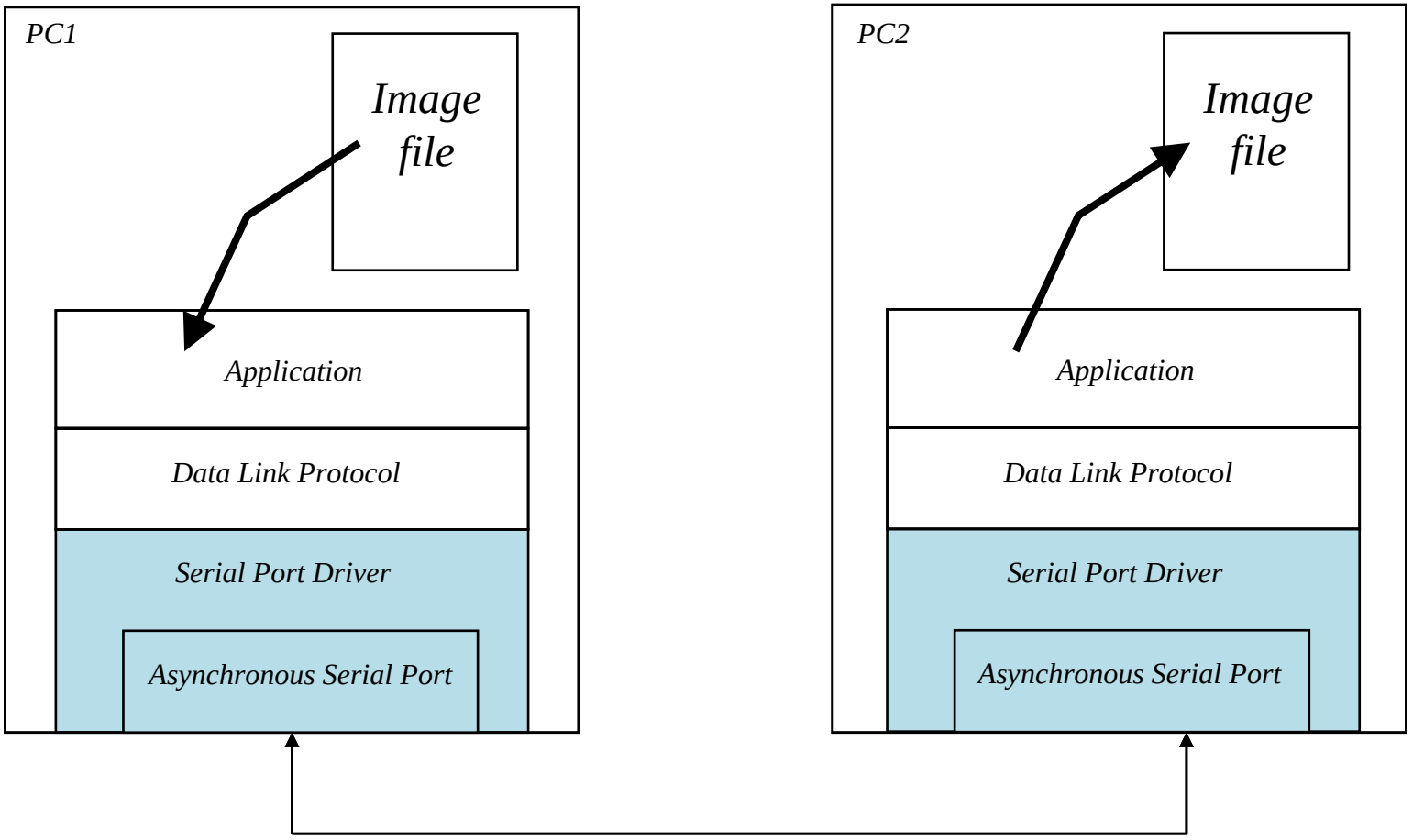
Communication System Overview



Data Link Protocols – Typical Functionality

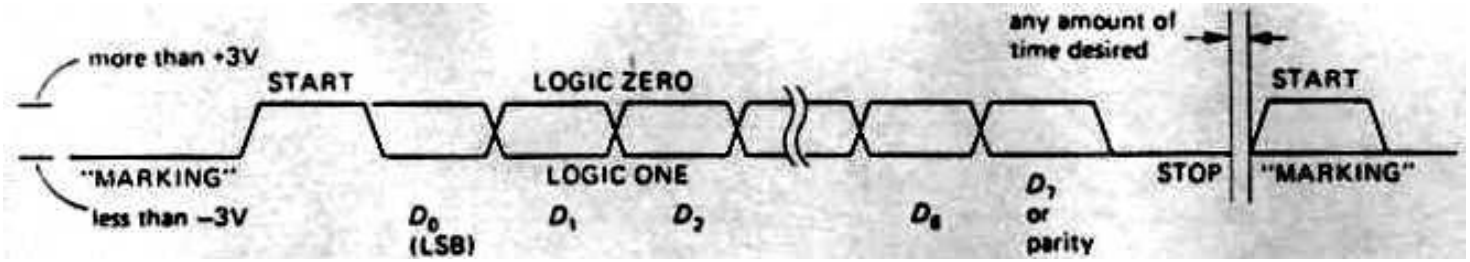
- Goal
  - Provide reliable communication between two systems connected by a communication medium – in this case, the serial cable
  - Service provided to applications using the communication medium
- Typical Functionality
  - Frame synchronisation – data organised in frames
    - Characters / flags to delimit start and end of frame
    - Data size may be implicit or indicated in the frame header
  - Connection establishment and termination
  - Frame numbering
    - Useful for frame acknowledgements, error control, flow control
  - Error control (e.g.: Stop-and-Wait, Go-back-N, Selective Repeat)
    - Acknowledgements after reception of a correct and ordered frame
    - Timers (time-out) – transmitter decides on retransmission
    - Negative acknowledgement (out of sequence frames) – receiver requests retransmission
    - Retransmissions may originate duplicates, which should be detected and eliminated
  - Flow control

Serial Port


PC1
Image file
Application
Data Link Protocol
Serial Port Driver
Asynchronous Serial Port
PC2
Image file
Application
Data Link Protocol
Serial Port Driver
Asynchronous Serial Port

Synchronous Serial Transmission

- Each character is delimited by
  - Start bit
  - Stop bit (typically 1 or 2)
- Each character consists of 8 bits (D0 – D7)
- Parity
  - Even – even number of 1s
  - Odd – uneven number of 1s
  - Inhibited (bit D7 used for data) – option adopted in this assignment
- Transmission rate: 300 to 115200 bit/s



RS-232 Signals

- Physical payer protocol between computer or terminal (DTE) and modem (DCE)
  - DTE (Data Terminal Equipment)
  - DCE (Data Circuit-Terminating Equipment)

TABLE 10.4. RS-232 SIGNALS

Name	Pin number 25-pin	Pin number 9-pin	Direction (DTE↔DCE)	Function (as seen by DTE)	
TD	2	3	→	transmitted data	} data pair
RD	3	2	←	received data	
RTS	4	7	→	request to send (= DTE ready)	} handshake pair
CTS	5	8	←	clear to send (= DCE ready)	
DTR	20	4	→	data terminal ready	} handshake pair
DSR	6	6	←	data set ready	
DCD	8	1	←	data carrier detect	} enable DTE input
RI	22	9	←	ring indicator	
FG	1	–		frame ground (= chassis)	
SG	7	5		signal ground	

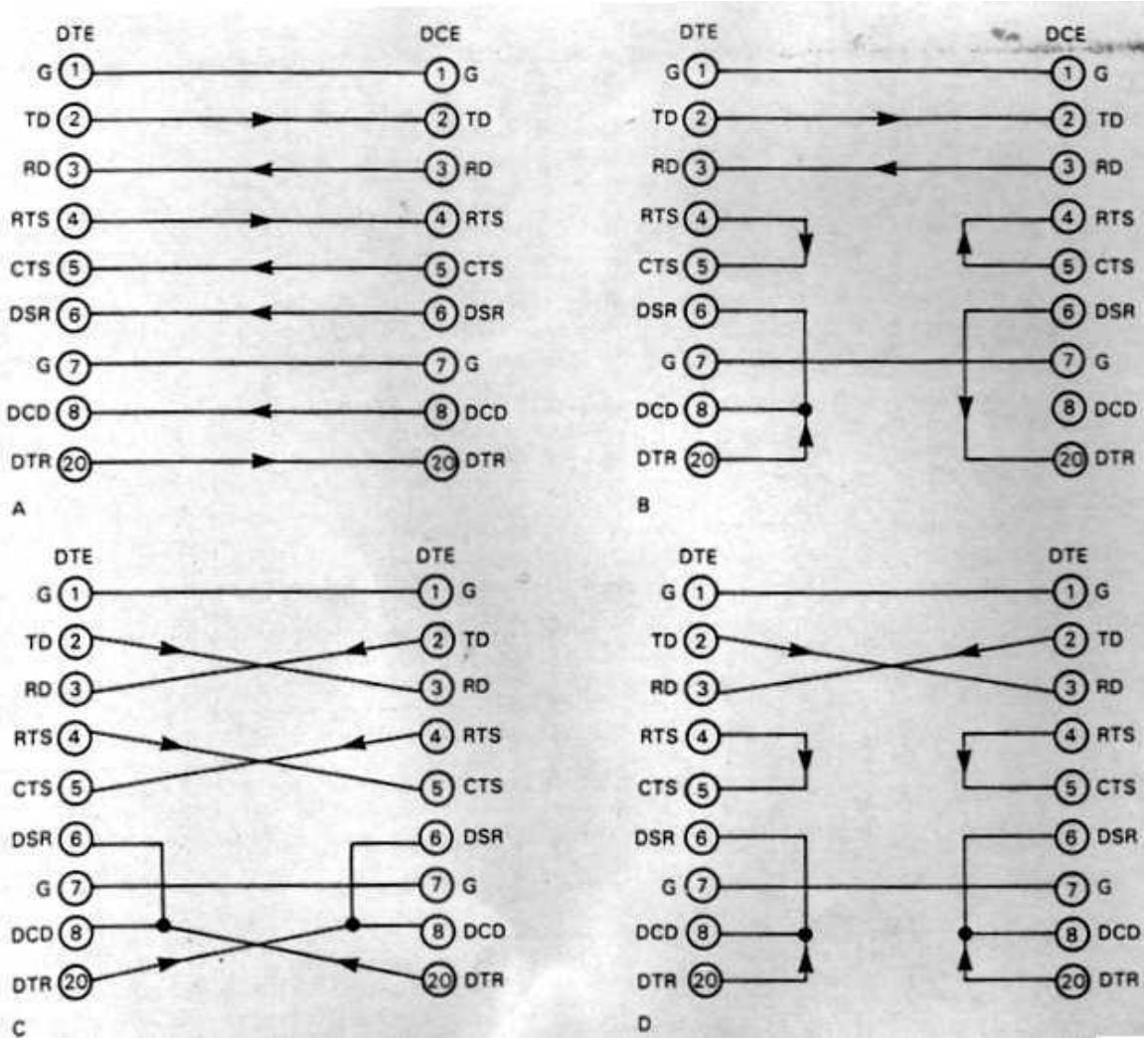
RS-232 Signals

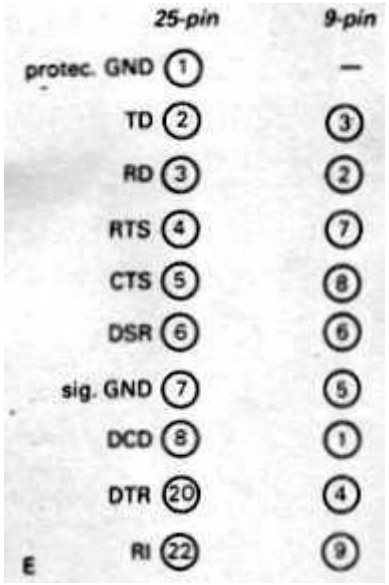
- **Active signal**
  - Control signal (> + 3 V)
  - Data signal (< - 3 V)
- **DTR (Data Terminal Ready) –** Computer on
- **DSR (Data Set Ready) –** Modem on
- **DCD (Data Carrier Detected) –** Modem detects carrier on phone line
- **RI (Ring Indicator) –** Modem detects ringing
- **RTS (Request to Send) –** Computer ready to communicate
- **CTS (Clear To Send) –** Modem ready to send
- **TD (Transmit data) –** Data transmission
- **RD (Receive data) –** Data reception

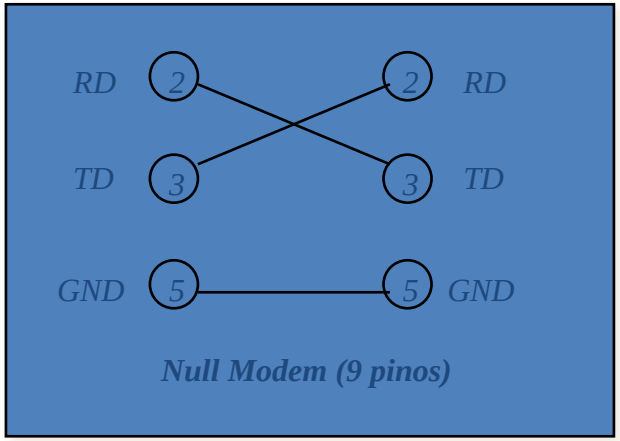
DB9

DB25

Equipment Connections



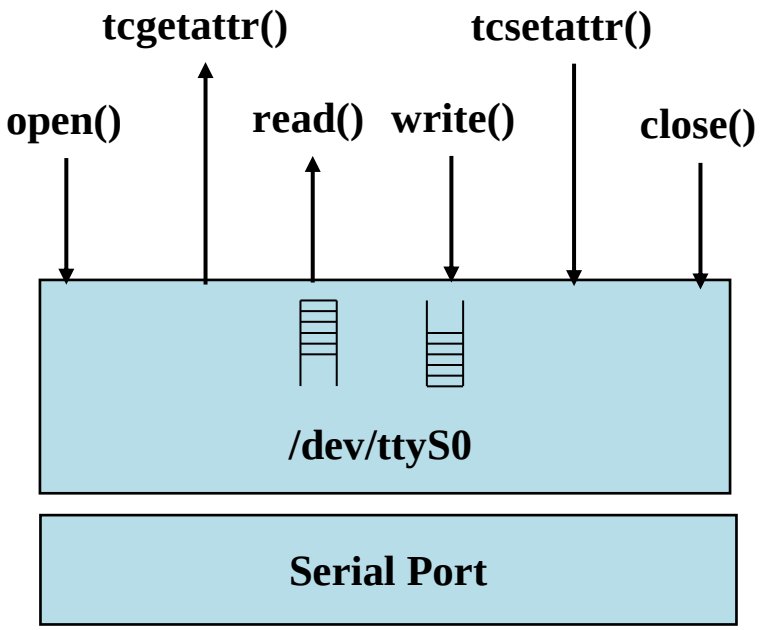




Unix Drivers

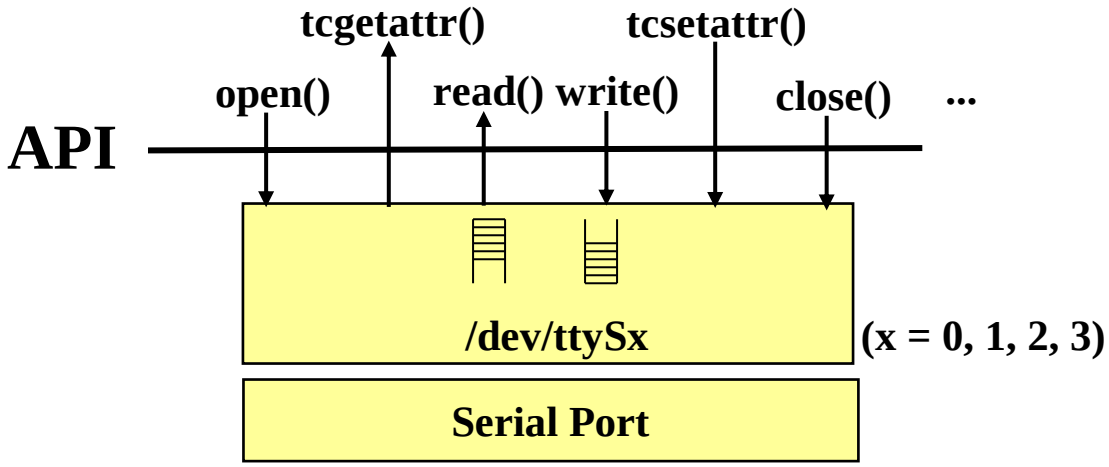
- Software that manages a hardware controller
  - Low level routines with privileged access
  - Reside in memory (part of the kernel)
  - Have associated hardware interrupt
- Access method
  - Mapped into Unix file system (/dev/hda1, /dev/ttyS0)
  - Services are similar to files (open, close, read, write)
- Driver types
  - Character
    - Read and write carried out multiples of a character
    - Direct access (data is not stored in buffers)
  - Block
    - Read / write as multiples of a block of octets
    - Data stored in random access buffer
  - Network
    - Read and write variable size data packets
    - Socket interface

Serial Port Driver – API



Serial Port Driver API

API – ***A****pplication* ***P****rogramming* ***I****nterface*



Some API Functions

int **open** (DEVICE, O_RDWR); /* returns a file descriptor */
int **read** (int fileDescriptor, char *buffer, int nChars); /* returns num characters read */
int **write** (int fileDescriptor, char *buffer, int nChars); /* returns num characters written */
int **close** (int fileDescriptor);

int **tcgetattr** (int fileDescriptor, struct termios *termios_p);
int **tcflush** (int fileDescriptor, int queueSelector); /* TCIFLUSH, TCOFLUSH or TCIOFLUSH */
int **tcsetattr** (int fileDescriptor, int mode, struct termios *termios_p); /* TCSANOW, TCSADRAIN or TCSAFLUSH */

Serial Port Driver API

- termios data structure: enables configuring and saving all serial port parameters

```
struct termios {
    tcflag_t  c_iflag;    /* Reception configuration flags */
    tcflag_t  c_oflag;    /* Transmission configuration flags */
    tcflag_t  c_cflag;    /* Control configuration flags */
    tcflag_t  c_lflag;    /* Local configuration flags */
    cc_t      c_cc[NCCS]  /* Control characters; NCCS = 19 */
    cc_t      c_line;     /* Not used */
};
```

- Man page: https://man7.org/linux/man-pages/man3/termios.3.html

Serial Port Driver API

- Example:

```
#define BAUDRATE B38400
struct termios newtio;

/* CS8:     8n1 (8 bits, no parity bit, 1 stop bit) */
/* CLOCAL:  Local link, no modem */
/* CREAD:   Enable characters reception */
newtio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;

/* IGNPAR:  Ignore parity errors */
/* ICRNL:   Convert CR to NL */
newtio.c_iflag = IGNPAR | ICRNL;

/* Non-processed output */
newtio.c_oflag = 0;

/* ICANON: Enable canonical input, disable echo and do not send signals to the
   program */
newtio.c_lflag = ICANON;
```

Serial Port Reception Types

- Canonic
  - read( ) returns only full lines (ended by ASCII LF, EOF, EOL)
  - Used for terminals
- Non-canonic
  - read( ) returns up to a maximum number of characters
  - Enables configuration of maximum number of characters
  - Adequate for reading groups of characters
- Asynchronous
  - read( ) returns immediately and send a signal to the application on return
  - Requires the use of a signal handler

Code Examples

Canonic Reception

```
int main(int argc, char *argv[])
{
  int fd, c, res;
  struct termios oldtio, newtio;
  char buf[255];

  fd = open(/dev/ttyS1, O_RDONLY | O_NOCTTY);
  tcgetattr(fd, &oldtio);

  bzero(&newtio, sizeof(newtio));
  newtio.c_cflag = B38400 | CS8 | CLOCAL | CREAD;
  newtio.c_iflag = IGNPAR | ICRNL;
  newtio.c_oflag = 0;
  newtio.c_lflag = ICANON;
  tcflush(fd, TCIFLUSH);
  tcsetattr(fd, TCSANOW, &newtio);

  res = read(fd, buf, 255);

  tcsetattr(fd, TCSANOW, &oldtio);
  close(fd);

  return 0;
}
```

Non-canonic Reception

```
int main(int argc, char *argv[])
{
  int fd, c, res;
  struct termios oldtio, newtio;
  char buf[255];

  fd = open(argv[1], O_RDWR | O_NOCTTY );
  tcgetattr(fd, &oldtio);

  bzero(&newtio, sizeof(newtio));
  newtio.c_cflag = B38400 | CS8 | CLOCAL | CREAD;
  newtio.c_iflag = IGNPAR;
  newtio.c_oflag = 0;
  newtio.c_lflag = 0;
  newtio.c_cc[VTIME] = 0;   /* Timeout between
                               characters */
  newtio.c_cc[VMIN] = 5;    /* Block until reading
                               5 characters */
  tcflush(fd, TCIFLUSH);
  tcsetattr(fd, TCSANOW, &newtio);

  res = read(fd, buf, 255); /* At least 5
                               characters */

  tcsetattr(fd, TCSANOW, &oldtio);
  close(fd);

  return 0;
}
```

Code Examples

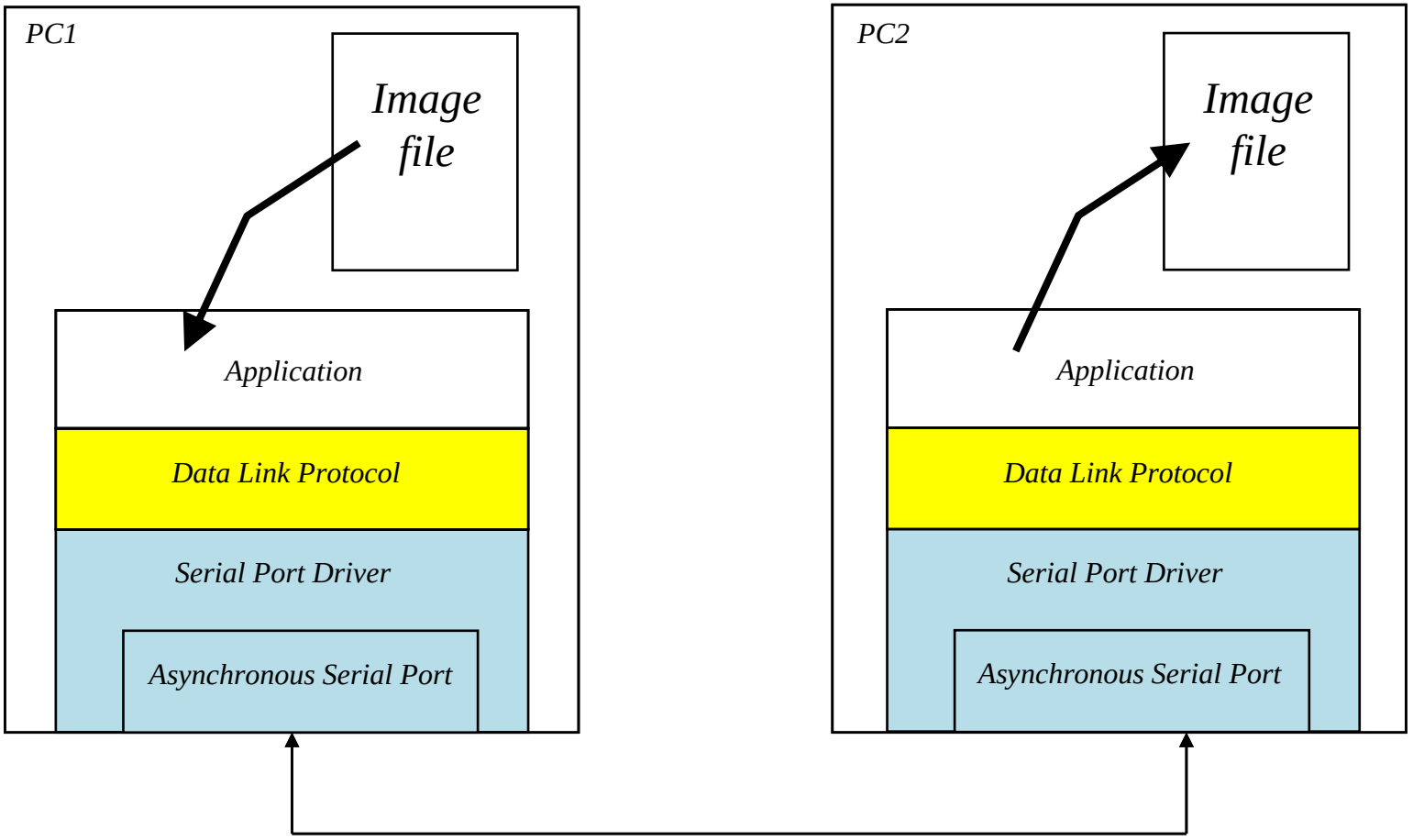
Asynchronous Reception

```
/* Signal handler */
void signal_handler_IO (int status) {
  wait_flag = FALSE;
}

int main() {
  /* Declare variables and open serial port */
  /* (…) */

  /* Configure async reception */
  saio.sa_handler = signal_handler_IO;
  saio.sa_flags = 0;
  saio.sa_restorer = NULL;  /* Obsolete */
  sigaction(SIGIO, &saio, NULL);
  fcntl(fd, F_SETOWN, getpid());
  fcntl(fd, F_SETFL, FASYNC);

  /* Configure serial port using termios struct */
  /* (…) */

  while (loop) {
    write(1, ".", 1); usleep(100000);
    /* After receiving SIGIO, wait_flag = FALSE, input
       is available to read */
    if (wait_flag == FALSE) {
      read(fd, buf, 255);
      wait_flag = TRUE; /* Wait for new input */
    }
  }
  /* Restore old port settings and close it */
  /* (…) */
  return 0;
}
```

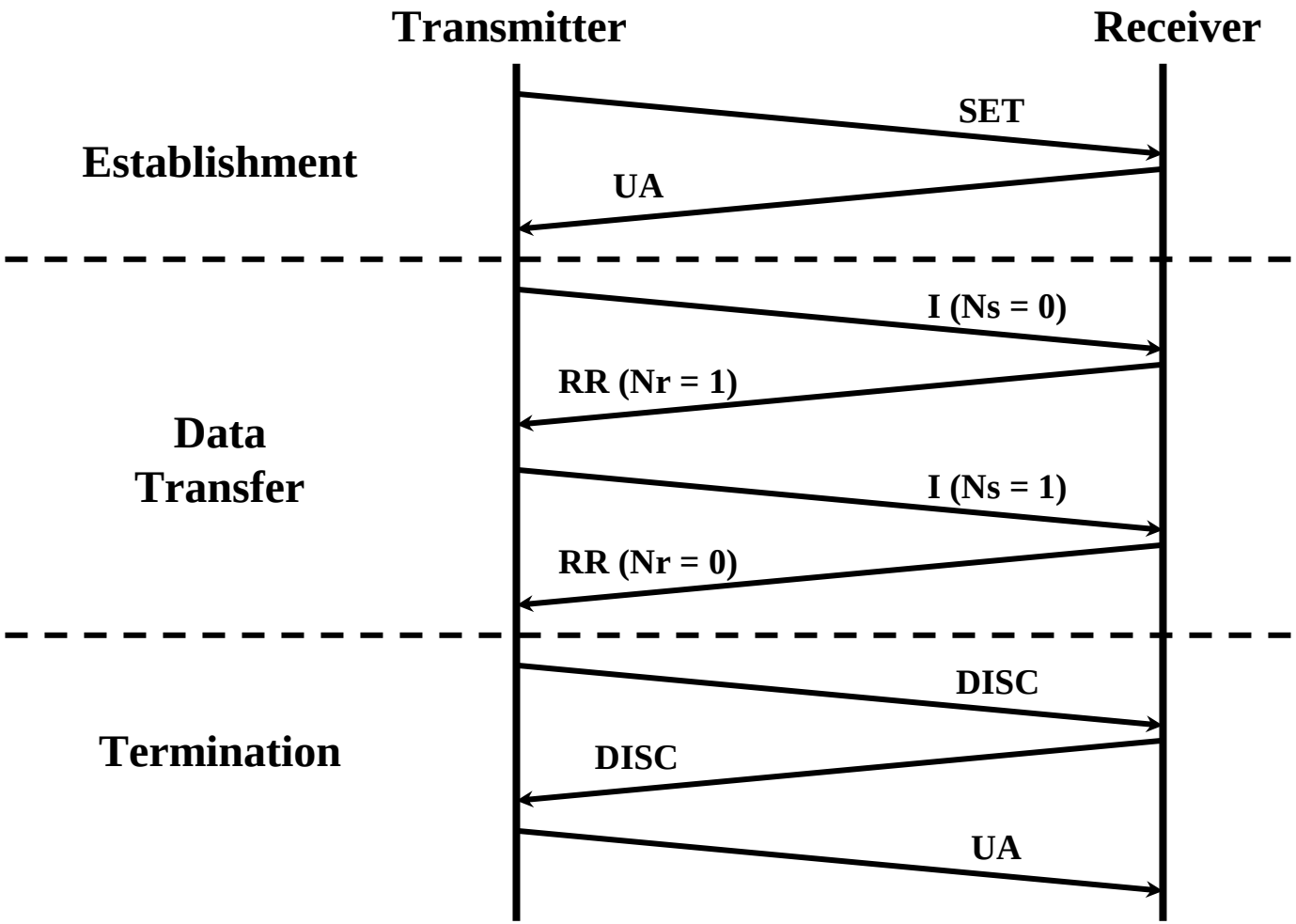
Multiple Reception

```
int main() {
  int fd1, fd2;  /* Input sources 1 and 2 */
  fd_set readfs; /* File descriptor set */
  int maxfd, loop = TRUE;

  /* open_input_source opens a device, sets the
port correctly, and returns a file descriptor */
  fd1 = open_input_source("/dev/ttyS1"); /* COM2 */
  fd2 = open_input_source("/dev/ttyS2"); /* COM3 */
  maxfd = MAX (fd1, fd2) + 1;  /* Max bit entry
                                  (fd) to test */
  while (loop) { /* loop for input */
    FD_SET(fd1, &readfs);  /* Set testing for
                              source 1 */
    FD_SET(fd2, &readfs);  /* Set testing for
                              source 2 */
    /* Block until input becomes available */
    select(maxfd, &readfs, NULL, NULL, NULL);
    /* Input from source 1 available */
    if (FD_ISSET(fd1))
      handle_input_from_source1();
    /* Input from source 2 available */
    if (FD_ISSET(fd2))
      handle_input_from_source2();
  }

  /* Restore old port settings and close them */
  /* (…) */
  return 0;
}
```

Data Link Protocol

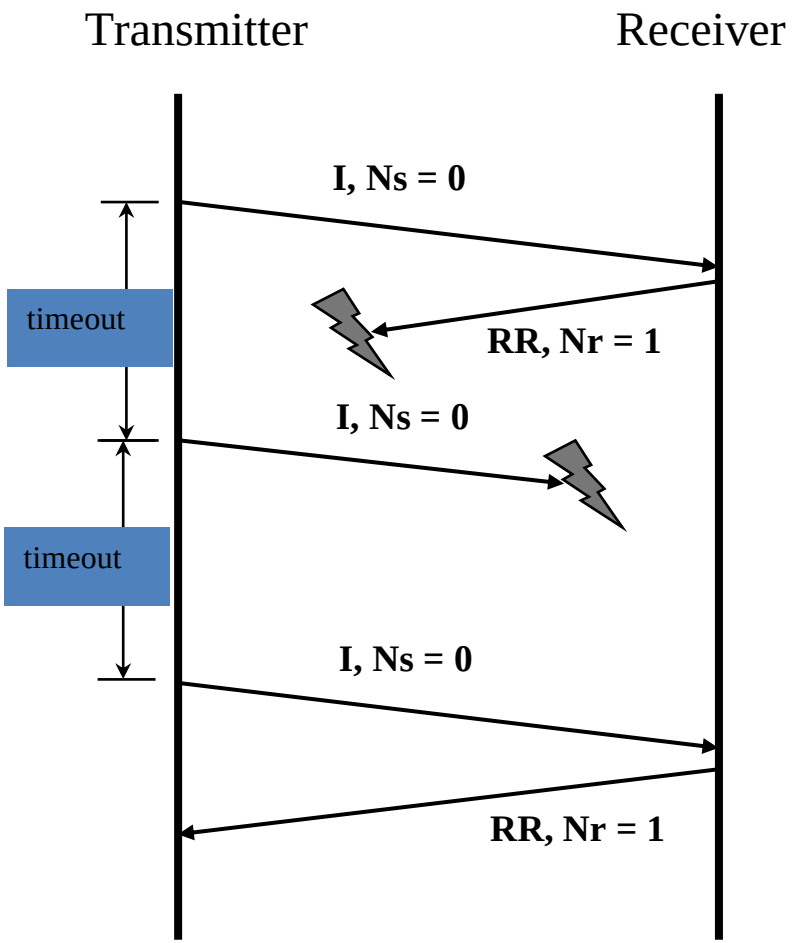


Data Link Protocol Phases



Data Transfer – Retransmissions

- Acknowledgement/ Error Control
  - *Stop-and-Wait*
- Timer
  - Set after an I, SET or DISC frame
  - De-activated after a valid acknowledgement
  - If exceeded (*time-out*), forces retransmission
- I frame retransmissions
  - After *time-out*, due to loss of frame or acknowledgement
    - Configurable maximum number of retrials
  - After negative acknowledgement (REJ)
- Frame protection
  - Generation and verification of the protection fields (BCC)



Data Link Protocol – Specification

- Frame delimitation is obtained by a special 8 bit sequence (FLAG)
  - Transparency must be guaranteed through bit stuffing
- Transparent data transmission, i.e., independent of the code used for transmission
- Transmission organised into 3 types of frames – Information (I), Supervision (S) e Unnumbered (U)
  - Frames have a header with common format
  - Only I frames have a field for data communication that carries application data without interpreting it – **layer independence / transparency**
- Frames are protected by an error correction code
  - In S and U frames, there is simple frame protection, since they do not carry data
  - In I frames, there is double independent protection for header and data fields, enabling the use of the header even if there are errors in the data field
- Stop and Wait error control, unit window and modulo 2 numbering

Specification Frame Reception

- I, S or U frames with a wrong header are ignored without action
- The data field of an I frame is protected by an own BCC
  - Even parity on each bit of the data and BCC
- I frames received without errors on the header and data field are accepted
  - If it is a new frame, the data field is passed to the application and the frame is confirmed with RR
  - If it is a duplicate, the data field is discarded, but the frame is confirmed with RR anyway
- I frames without detected errors on the header but with errors detected on the data field: data field is discarded, but control field can be used to trigger an action
  - If it is a new frame, a retransmission request can be issued with a REJ request, triggering a faster retransmission than waiting for a timeout
  - If it is a duplicate, the frame should be confirmed with RR
- I, SET e DISC frames are protected by a timer
  - If a time-out occurs, transmission should be repeated a maximum number of times, e.g., 3, which should be configurable

Frames – Header Delimitation

- All frames are delimited by flags (**01111110**)
- A frame may be initialised with one or more flags, and this must be taken into consideration by the reception mechanism
- I, SET e DISC frames are designated Commands and UA, RR and REJ frames Replies
- All frames have a header with a common format
  - A (Address field)
    - **00000101** (0x05) in Commands sent by the Transmitter and Answers sent by the Receiver
    - **00000010** (0x02) in Commands sent by the Receiver and Answers sent by the Transmitter
  - C (Control fields) – Defines the type of frame and carries the sequence numbers N(s) in I frames and N(r) in S frames (RR, REJ)
  - BCC (Block Check Character) – Provides error control based on an octet that guarantees that there is an even pair of 1's (even parity) for each bit position, considering all octets protected by the BCC (header or data) and the BCC (before stuffing)

Specification – Frame Formats

- Information Frames

(before *stuffing* and after *destuffing*)

F **Flag**
A **Address field**
C **Control field** **0 0 0 0 0 0 S 0** **S = N(s)**
D_1 ... D_N **Data field (contains data generated by the application)**
$BCC_{1,2}$ **Independent protection fields (1 – header, 2 – data)**

- Supervision and Unnumbered Frames

F **Flag**
A **Address field**
C **Control field**

SET (set up)	**0 0 0 0 0 0 1 1**
DISC (disconnect)	**0 0 0 0 1 0 1 1**
UA (unnumbered acknowledgment)	**0 0 0 0 0 1 1 1**
RR (receiver ready / positive ACK)	**0 0 R 0 0 0 0 1**
REJ (reject / negative ACK)	**0 0 R 0 0 1 0 1**

R = N(r)

BCC_1 **Protection field (header)**

Transparency – Why?

- This work uses asynchronous communication
  - This technique is characterised by the transmission of characters (short sequence of bits, whose number can be configured) delimited by a Start and a Stop bit
- Some protocols use characters (words) of a code (e.g., ASCII) to delimit and identify the frame fields and support protocol mechanisms
- For transparent communication, i.e., communication independent of the code used for transmission, it is necessary to use escape mechanisms
  - To identify the occurrence of delimiting characters in the data and replace them so that they can be correctly interpreted at the receiver

Transparency – Why?

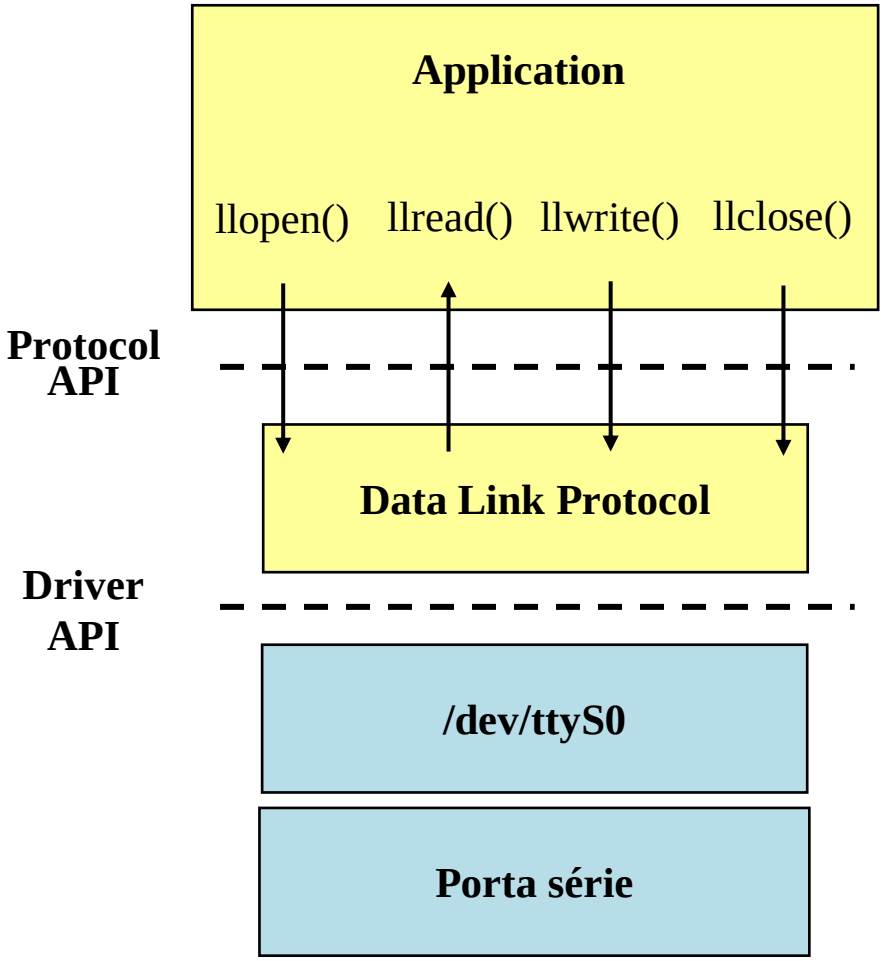
- The protocol to be implemented here is not based on any code, so the transmitted / received characters should be interpreted as plain octets (bytes), where any of the 256 possible combinations can occur
- To avoid that a character inside a frame is wrongfully recognised as a delimiting flag, you need to use a mechanism that provides transparency
  - You shall use a mechanism called bit stuffing, used in common link protocols like PPP or HDLC
  - Your protocol shall use the same mechanism used in PPP, using the escape byte **01111101** (**0x7d**)

Transparency – Byte Stuffing

- If the octet 01111110 (0x7e) occurs inside a frame, i.e., the pattern corresponding to a flag, the octet is replaced by the sequence 0x7d 0x5e (escape octet followed by the result of the exclusive or of the replaced octet with octet 0x20)
- If the octet 01111101 (0x7d) occurs inside a frame, i.e. The escape octet pattern, the octet is replaced by the sequence 0x7d 0x5d (escape octet followed by the result of the exclusive or of the replaced octet with octet 0x20)
- The generation of BCC considers only the original octets (before stuffing), even if any octet must be replaced by the escape sequence
- The BCC verification is performed on the original octets, i.e., after the inverse operation (destuffing) in case there had been any substitution of the special octets by the escape sequence

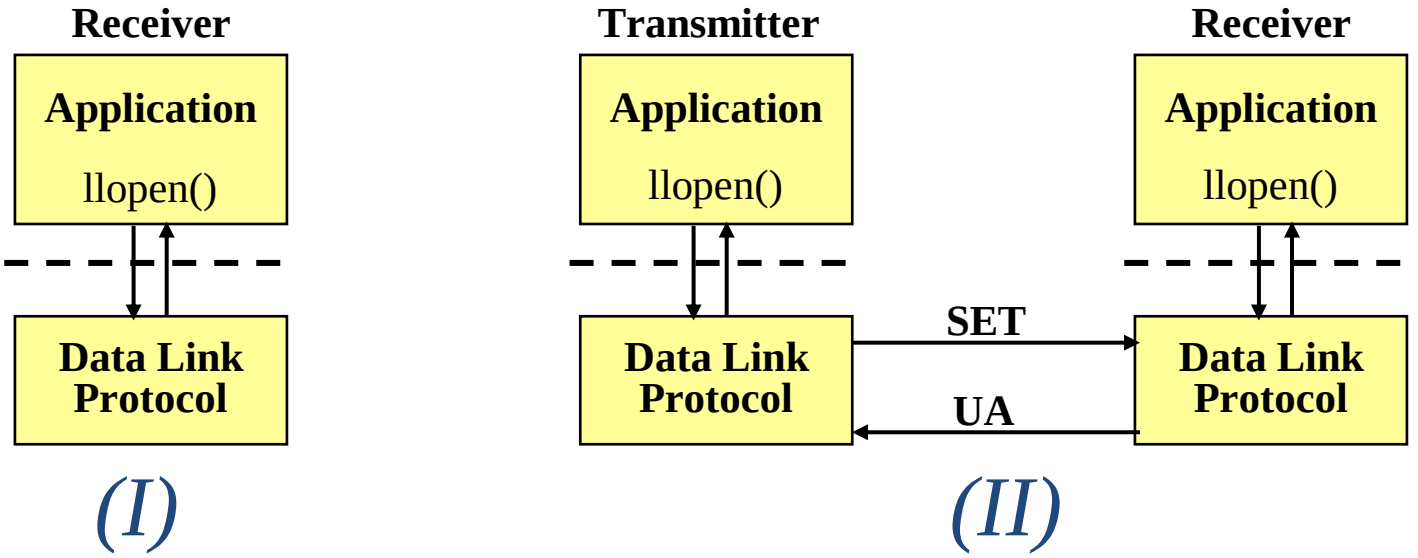
Protocol AP Interface

- Implement the protocol API in a .h file to be included by the applications
- The application should only use the functions in that file



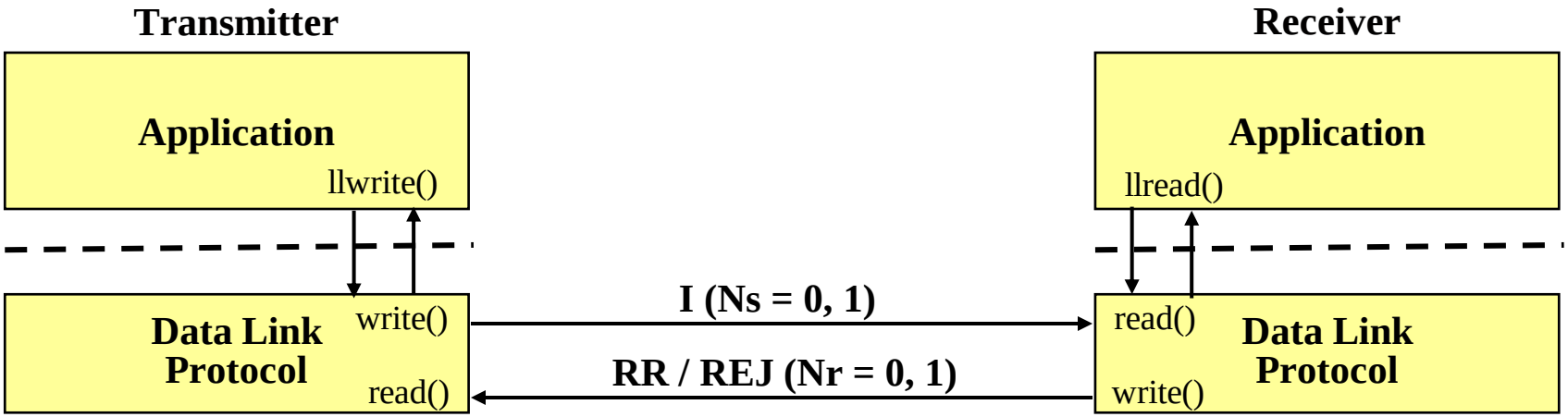
Protocol API – open

- int llopen(linkLayer connectionParameters)
  - Arguments
    - connectionParameters: Connection parameters, including:
      - port: COM1, COM2, ...
      - role: TRANSMITTER / RECEIVER
  - Return value
    - ”1” on success
    - “-1” on failure / error



Protocol API – read / write

- int llwrite(char *buf, int bufSize)
  - Arguments
    - buf: Array of char to transmit
    - bufSize: Length of the char array
  - Return value
    - Number of characters written
    - "-1" in case of failure / error
- int llread(char *packet)
  - Arguments
    - packet: Char array with the received packet
  - Return value
    - Array length (number of chars read)
    - "-1" in case of failure / error



Protocol API

- Example data structures
  - Application

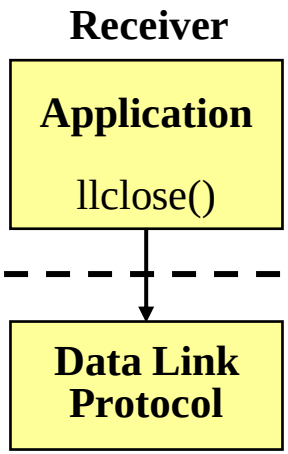
```
struct applicationLayer {
  int fileDescriptor;  /* File descriptor corresponding to serial
                          port */
  int status;          /* TRANSMITTER | RECEIVER */
}
```

  - Protocol

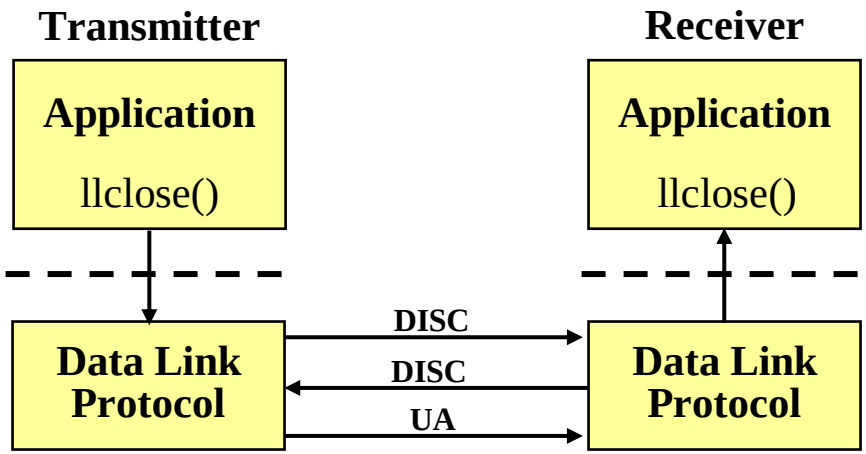
```
struct linkLayer {
  char port[20];                  /* Device /dev/ttySx, x = 0, 1 */
  int baudRate;                   /* Transmission speed */
  unsigned int sequenceNumber;    /* Frame sequence number: 0, 1 */
  unsigned int timeout;           /* Timeout value: 1 s */
  unsigned int numTransmissions; /* Number of retransmissions
                                    in case of failures */
  char frame[MAX_SIZE];           /* Frame (char array) */
}
```

Protocol API – close

- int llclose(int showStatistics)
  - Arguments
    - showStatistics: Flag indicating if the program should print statistics to the console
  - Return value
    - “1” on success
    - “-1” in case of failure / error

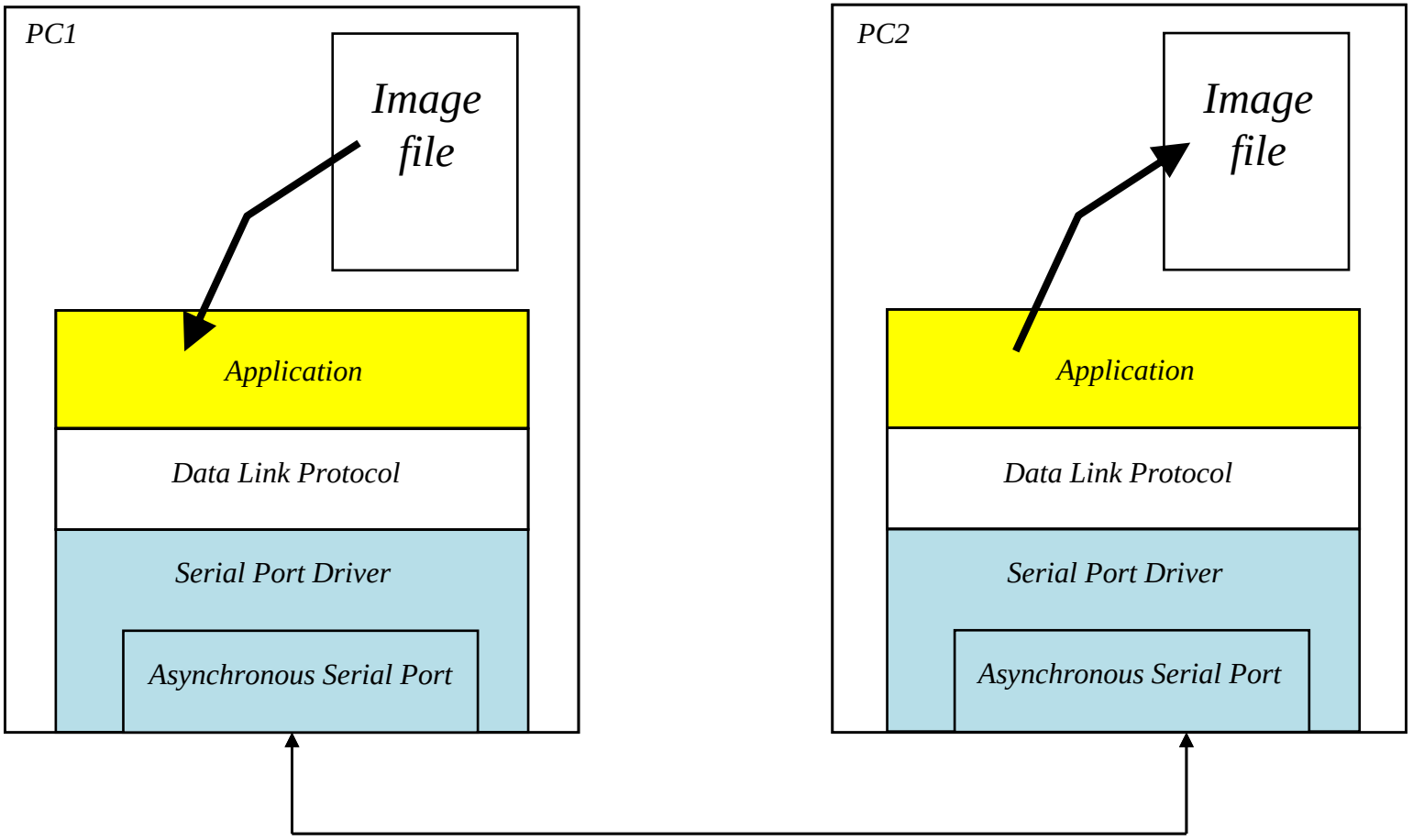


(I)



(II)

Application



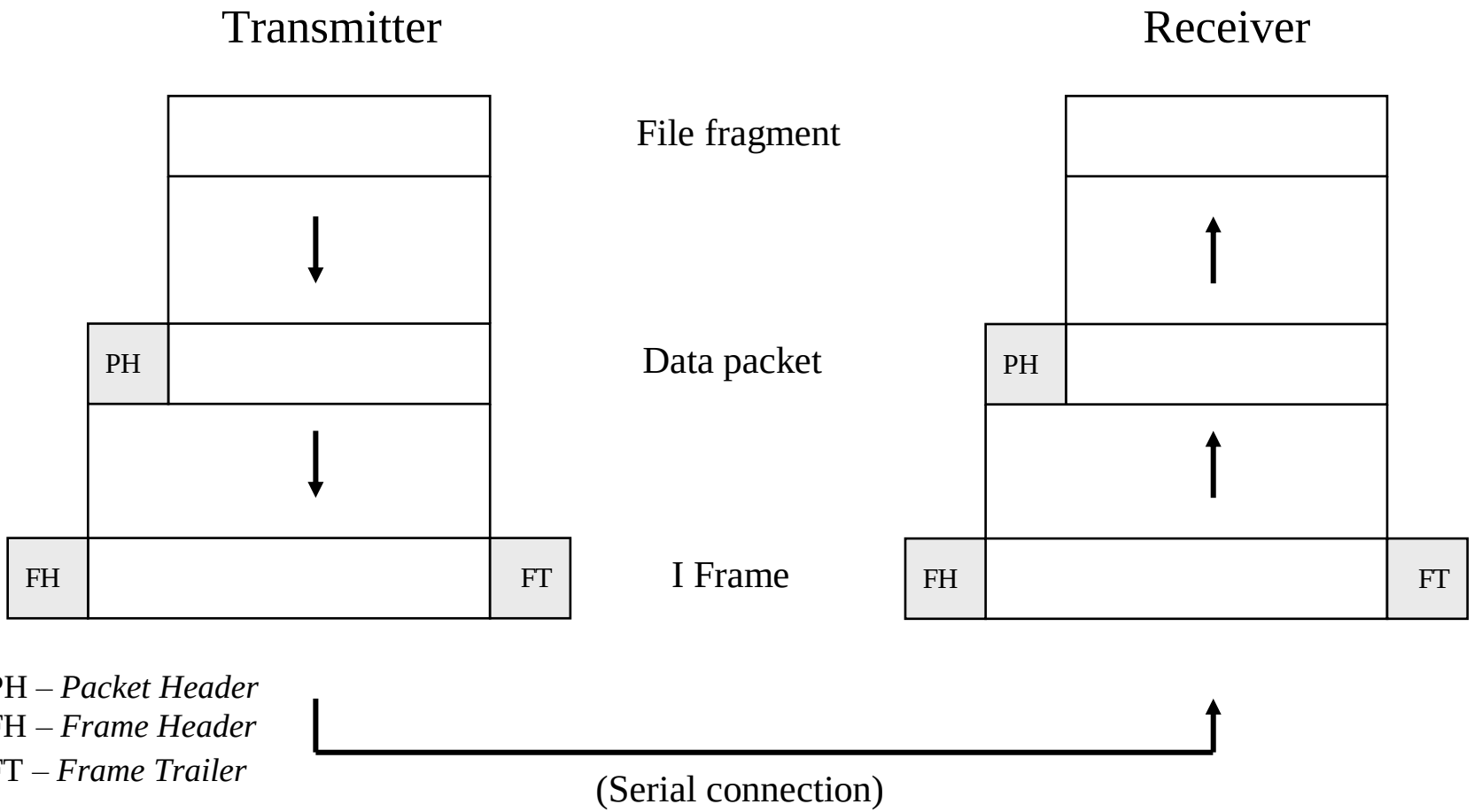
Application

- Simple file transfer application
  - Application uses its own protocol
- Application uses two types of packets
  - Packets that signal the start and end of file
  - Packets that contain file fragments

Application Packets and Data Link Frames

- The file to be transmitted is fragmented – the fragments are encapsulated into data packets and these are carried in the data fields of the I frames of the data link protocol
  - Besides data packets, the application protocol uses control packets
  - Application packet format is defined ahead
- Transmitter is the machine that sends the file and Receiver is the machine that receives the file
  - Only Transmitter sends application packets (data or control)
- Both transmitter send and receive frames to implement the data link protocol

Data Packets and I Frames



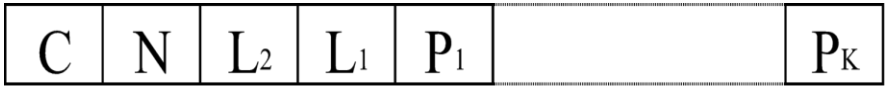
Control packets are also carried in I frames

Test Application – Specification

- The goal is to develop a simple file transfer application that uses the reliable transfer service of the data link protocol
- The application transmitter can send two types of packet
  - Control packet to signal the start and end of the file transfer
  - Data packets containing the actual file fragments
- The control packet that signals the beginning of transmission (start) must have a field indicating the file size and an optional field for the file name (and eventually other fields)
- The control packet that signals the end of transmission should repeat the information in the start packet
- Data packets must have a 1-octet field with a sequence number and another 2-octet field indicating the size of the data field
  - This size depends on the maximum packet size of the information frames (I frames)
  - These fields allow additional verification of data integrity
  - The number of I frames is independent of the sequence numbers

Application Layer Packets

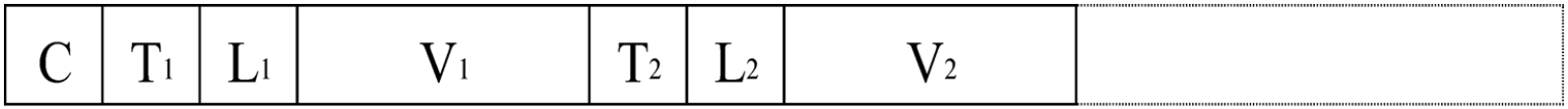
- Data packet



  - C – control field (value: 0 – data)
  - N – sequence number (mod 255)
  - L2 L1 – number of octets (K) in the data field
  - P1 ... PK – data field (K octets)

$(K = 256 * L_2 + L_1)$

- Control packet



  - C – control field (values: 1 – start; 3 – end)
  - Each parameter (size, file name or other) is coded as TLV (Type, Length, Value)
    - T (one octet) – identifies the parameter (0 – file size, 1 – file name, other values – to be defined if necessary)
    - L (one octet) – indicates the size in octets of the field V (parameter value)
    - V (number of octets indicated in L) – parameter value

Layer Independence

- Layered architectures are based on layer independence
- This has following implications in this assignment
  - In the data link layer, no processing is done on the headers of the packets to be carried by I frames – this information is considered inaccessible to the data link protocol
    - Athe the data link level there is no distinction between application control and data packets, nor is the numbering taken into account
  - The application does not know details of the data link protocol, only how to access its service
    - The application protocol does not know the frame structure or the delimitation mechanism, the existence of stuffing, the frame protection mechanism, eventual retransmissions, etc
    - All these functions are implemented exclusively at the data link layer
  - In particular, the I frame numbering and the application data packet numbering should be completely independent and unrelated

Data Link Layer Protocol Evaluation

- The data link layer protocol will be tested and evaluated using two applications:
  - The application developed by each group (optional)
  - An application provided to all groups that uses the defined link layer protocol interface
    - This application allows the evaluation of the layer independence principle
    - You shall include the .h file and compile the application code, and run tests to verify protocol operation
    - You shall not change the source code of the application