433-353 Networks and Communications
Project 1 : Ethernet frame format and CRC calculations

Department of Computer Science and Software Engineering
University of Melbourne

1 Introduction

A simple network consists of *hosts* connected via a *hub*. Each host receives a copy of what ever data is transmitted — even the transmitting host receives a copy (this helps to detect collisions). This is depicted in Fig. 1.

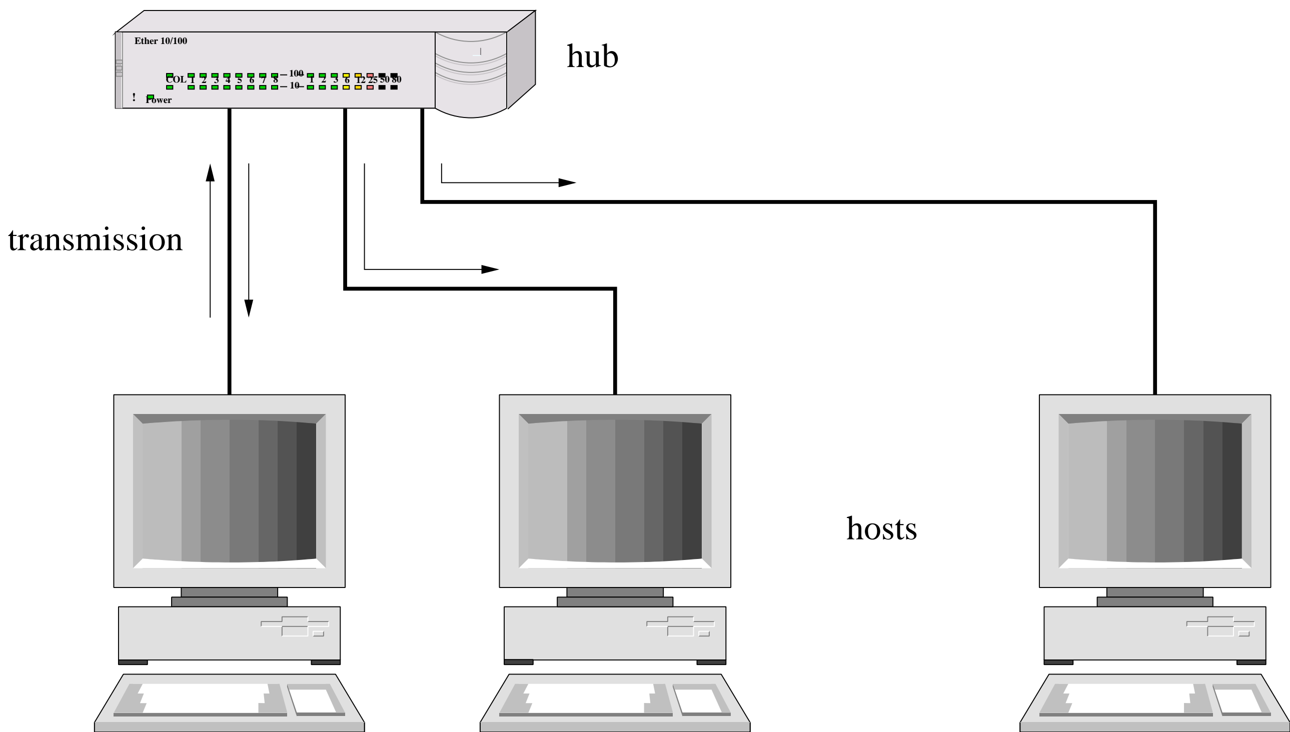

Figure 1: Simple network using a hub.

A common Data-link Layer protocol for transmitting over a shared medium, such as provided by a hub, is *Ethernet*. Fig. 2 shows the frame format for an Ethernet frame. The *preamble* is a sequence

preamble 7 bytes	SOF 1 byte	destination address 6 bytes	source address 6 bytes	frame length 2 bytes	data 46 – 1500 bytes	CRC 4 bytes

Figure 2: The format of an Ethernet frame. SOF is Start Of Frame byte.

of seven `0xAA` bytes and is followed by a *start of frame* (SOF) byte, `0xAB`. Technically the preamble and SOF are not part of the frame, but are required to help the receiver detect the start of a frame. In this project, the Data-link Layer will be responsible for writing the required preamble and SOF and for detecting and removing it from received data.

2 Virtual hub and hosts

In this project we simulate a hub using a *virtual hub* program and simulate hosts using a *virtual host* program. The virtual hub is a server and the virtual hosts are clients, however you need not understand client/server programming for this project as all of the details concerning this are hidden in the Physical Layer module. You need only worry about the format and content of the data that is being transmitted. The project files are available at `/home/subjects/353/local/Project1` and includes a `Makefile`. Make of copy of these files and build the system. You may need to change the Makefile to omit/include the libraries. As is, it should compile on the Department servers without trouble.

To use the system, the virtual hub or *vhub* is executed and becomes a daemon:

```
prompt> ./vhub
listening on localhost 54667
pid 13669
prompt>
```

Note that the vhub reports `listening on localhost 54667` which means that it is running on the localhost (the machine that you are logged in to, e.g. `muntye.cs.mu.oz.au`) and can be connected to on *port* number 54667. Record this number for future use. Since the vhub is running as a daemon, it should be killed when not needed, e.g. `kill 13669`. Use `ps -u username` to list all processes that you have on the machine. DON'T FORGET to kill your vhub when not needed. The vhub will not be killed by logging out — it will remain until you kill it.

A virtual host or *vhost* is not executed in the background. For this project you will need to use two terminal windows, one for each vhost. The other members of your group can run vhosts from their terminals, but only one vhub should be working per project group.

The vhost program requires the host name and port number of the vhub to which it will connect. For example:

```
prompt> ./vhost localhost 54667
Layer 1: connect to localhost
```

or from another machine:

```
prompt> ./vhost muntye.cs.mu.oz.au 54667
Layer 1: connect to muntye.cs.mu.oz.au
```

The vhost is actually a very simple program. It simply reads from the standard input and writes to the standard output. Any data that is read from the standard input is transmitted to the vhub. Any data that is received by the vhub is transmitted to all vhosts connected to it (at most 16 vhosts can connect to a single vhub). Any data received by a vhost from the vhub is written to standard output.

In the simplest sense, the vhost/vhub system can be used like a chat program! Try using it as a chat program with the members of your group. If you are working alone, open a second terminal window and connect a second vhost to the vhub.

Recall that the sender as well receives the data that is sent. Continuing the above example, typing `hello there!\n` would send the data which is received by the vhub and is also received by all vhosts. Closing standard input will terminate the vhost which sleeps for 3 seconds before actually returning.

```
prompt> ./vhost localhost 54667
Layer 1: connect to localhost
hello there!
hello there!
```

Typing something like `prompt> vhost localhost 54667 < file.txt` will transmit an entire file and then terminate. Now stop chatting.

3 The Layers of the Virtual Host

You need not understand the virtual hub, but you must understand the virtual host to complete your project. Fig. 3 shows the basic layer interfaces. In this project, Layer 1 and Layer 3 are given to

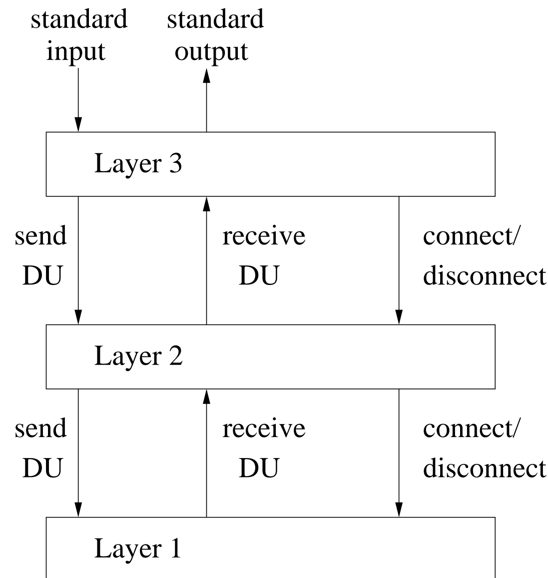

Figure 3: Layers of the virtual host.

you complete. However Layer 2 is incomplete. The given program will compile and work, but the data is not formatted correctly for Ethernet transmission, it is simply written raw. The code for connecting and for establishing the linkages between layers is there. But Layer 2 does not format the data received from Layer 3, neither does it do anything with the data received from Layer 1 apart from pass it directly to Layer 3. You must write the appropriate functions in Layer 2.

Layer 2 implements a skeleton `void L2_sendDU(L2_IDU l2idu)` function which is called by Layer 3 when Layer 3 wants to send data. Examine `layer2.h` to see the data type declarations. The maximum size of the DU is 1500 bytes.

```
#define L2_MTU 1500

typedef struct
  {
    unsigned char data[L2_MTU];
    unsigned int length;
  }
L2_IDU;
```

You must complete the implementation of the `L2_sendDU` function by framing the received data into an Ethernet frame as shown in Fig. 2 before sending to Layer 1. Layer 1 implements a similar function `L1_sendDU(L1_IDU l1idu)` where in this case the data type is simply an `unsigned char`. This means that the frame must be transmitted one character at a time.

To complete the project you must also complete the `void L2_receiveL1_IDU(L1_IDU l1idu)` function which receives a single byte at a time from Layer 1. These bytes are of course the same bytes which were sent (as in this ideal system there is no noise). The function needs to buffer the bytes until it receives an entire frame. The data should be extracted from frames that have no CRC error and then sent up to the Layer 3 using the function `void receiveL2_IDU(L2_IDU l2idu)` which links to Layer 3.

4 What you must do

The following implementation is required to be incorporated into the Layer 2 module:

1. Implement a function that constructs a frame, given a data unit from Layer 3. The `sendBuffer` has been provided to store this frame.
2. Implement a function to compute the CRC that is part of the transmitted frame. The CRC must be appended to the frame in the appropriate place.
3. Implement a function that will receive a frame, extract the DU and send it to Layer 3. It must be able to detect the length of the frame as the frame is received one byte at a time and the frame length is variable. The `receiveBuffer` has been provided to store this frame. A `tempBuffer` is also available.
4. Implement a function that will check the CRC of the received frames and report any errors.

4.1 Frame format and computing the CRC

Fig. 4 gives a detailed description of the frame that must be constructed. The data is given from Layer 3. The length L is clearly the length of the data and should be written using big-endian addressing (network order). The CRC should also be written using big-endian addressing. The CRC calculation includes everything apart from the preamble and SOF. The polynomial is shown and indicates that x^n is the most significant bit of `A[8]`. The destination and source address may be set to zero for this assignment. The polynomial for the CRC calculation should be as used in IEEE 802.3 CRC32:

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1.$$

If the above polynomial is used to generate a CRC for the character string `"ABC"` (with of course 32 bits appended to the end of this string), then the answer is `0xAF8229AA`. In other words, instead of computing the CRC for the bytes in the frame, compute the CRC for the bytes `A`, `B` and `C`. This can be used as a weak test for correctness of your CRC computation. You should note that this simple CRC calculation is not identical to that used for CRC32. In the real case, the bytes need to be reversed and the initial/final computation conditions are slightly different. However, using the method for computing a CRC as given in the lectures and textbook, you should be able to arrive at the CRC just given.

4.2 Frame check error

If a received frame is found to have a CRC error then your Layer 2 should print:

```
CRC CHECK ERROR
```

and *not* pass the data up to Layer 3. The error message should be printed to standard error. The program should continue receiving frames. It should only terminate when the standard input is closed (termination is handled by Layer 3, so you needn't worry about it).

4.3 Possible errors

Currently, there are no errors in the system (unless a real network error occurs!). Thus, we need to introduce errors into the frame in order to see that the error is detected. In this project you may assume that an error only occurs in the data region of the frame and that an error will only cause a bit to be inverted. For example, the length field will never be received in error and bits will not be "deleted" or "inserted".

eplacements

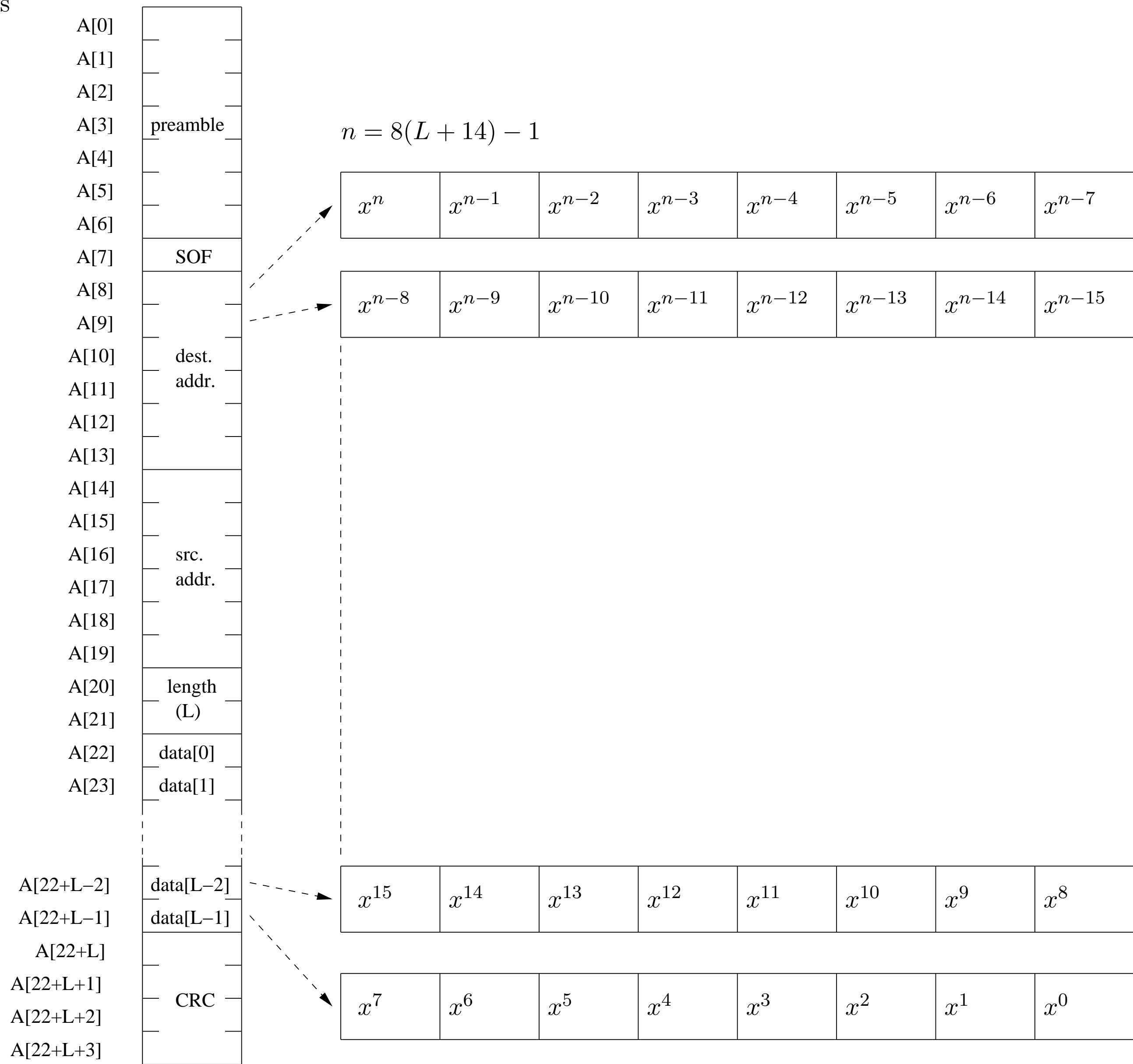

Figure 4: Detailed frame contents.

5 Administration

This project is worth 15% of your total mark for this subject. Roughly 5% is allocated for the project report, 5% for correct formatting and sending of a frame and 5% for correctly receiving and checking a frame for errors. The due date for this project is set to Monday, September 15th, $5pm$.

5.1 Groups

You may work in groups of up to 3. No more than 3 students may be in a single group. Each group must have exactly one *group leader*. The word document file
`/home/subjects/353/local/Project1/GroupInstructions.doc`
contains instructions on forming groups. Final assessment is based on the names given in the project report (one report per group). Each group member receives the same mark.

5.2 How to submit

You must submit a project report (report.txt) file and your modified layer2.c file. You should not modify or submit any other files:

```
prompt> submit 353 1 layer2.c report.txt
```

Your project report should be no longer than 2 pages. It should contain a complete description of all interesting variables that you needed to add to your program, all functions that you added and the general method that you used to compute the CRC (you may use pseudo-code explanation). Your report is also an opportunity to express any problems that you encountered and any assumptions that you needed to make. You need not worry about the efficiency of your algorithms. You need only ensure that your algorithms are correct. A layer2.c that doesn't compile or that has no changes will most likely receive no marks.

Any missing/extra details will be posted to the 353 news group and WWW pages. Be sure to check both these sources for updates concerning this project.