C64 Communications Program

In this article Nick Gammon describes his modem communication program for the Commodore 64. It is written in G-Pascal for the Commodore 64, and uses the Christensen protocol.

USING THE Christensen Protocol (described in *Your Computer* – May and June 1983) has several advantages – one of which is that it is already widely in use for data transmission. The protocol itself, and various implementations (such as YAM on CP/M systems), are in the public domain, making them readily available.

This program is directly compatible with the Mi-Computer Club (MiCC) bulletin board. Once you have typed in the program, you can directly access public domain software (if you are a member of MiCC) with minimum effort and maximum reliability.

You can also use it to converse with any other remote computer, have conversations between two Commodore 64 owners, or transfer programs between one Commodore 64 and another Commodore 64 or any other computer which has a program using the Christensen protocol.

What You Need

To use this program you will need:

a) A Commodore 64

b) An RS232 serial interface plugged into the user port (these are priced at about $50).

c) A modem connected to your telephone. (There was an article on modems in November 1983 *Your Computer*). You can use a 'direct coupled' or an 'acoustically coupled' modem. Modem prices vary; however, you could expect to get a cheap but satisfactory one for under $200.

d) A cable between the modem and the RS232 interface. As far as the Commodore 64 is concerned you only need to connect to pins 2, 3 and 7 (transmit data, receive data and ground).

e) A copy of G-Pascal – currently available for $79.50 from Commodore dealers.

Other Computers

If you don't have a Commodore 64, this program will not be of direct use to you. However, as it is written in Pascal it is relatively easy to follow – you should find the general methods used helpful in developing a similar program for your own computer.

Why Use A Protocol For Transferring Files?

While it is possible to write a simple 'dumb terminal' program in about ten lines of code, transferring files is a little more complicated. The reason for this is occasional noises on the telephone line may introduce errors, which might be acceptable if you are just having a conversation with someone at the other end of the line, but can cause irritating and hard-to-find errors if embedded in the middle of a program.

Data integrity (correct transmission of files) is not just 'handy', it is essential if you are to have any confidence in using your telephone for sending programs back and forth.

The Christensen protocol provides this integrity in a number of ways:

1. The sender and receiver 'synchronise' by using an agreed sequence of characters to start things rolling. This provides proper synchronisation even if the sender and receiver request transmission at different times (within no more than 60 seconds of each other).

2. Data is broken into 128-byte blocks so that if an error occurs it is only necessary to re-transmit 128 bytes, not the whole file.

3. Each block is numbered to ensure data is received in the correct sequence.

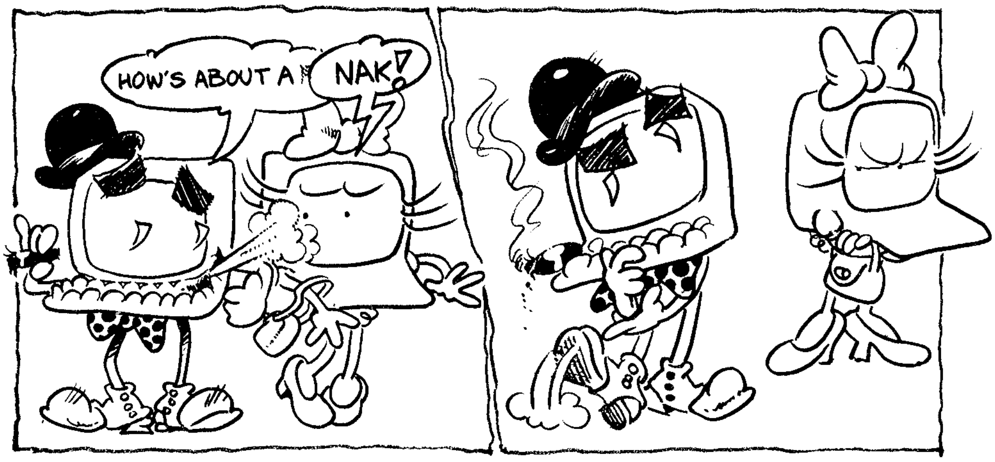
4. Each block has a sum check (optionally a cyclic redundancy check), to confirm that the data in that block is correct.

5. The program has provision for handling 'timeouts' – in other words, if no data at all is received within a predetermined time, the sending end re-transmits the block so that the program doesn't 'hang' indefinitely.

6. The program also performs a cyclic redundancy check on the whole file (as well as on individual blocks), to further ensure that the file was transmitted correctly.

Cyclic Redundancy Checks

The program uses cyclic redundancy checking for ensuring the integrity of both individual blocks of transmitted data and the whole file. A cyclic redundancy check (CRCK for short) is an enhanced method of doing a 'sum check' on a block of data. A sum check is performed by adding up each byte of data and retaining the low-order byte. A CRCK is performed in a more complicated way: in fact, there are various CRCK algorithms. The modem program uses two different methods in order to be compatible with YAM. Both methods involve calculating a two-byte result, by shifting the previous result left one bit and adding in the new bit (or byte), to provide the new result. However, unlike a simple sum check, the CRCK routines have provision for not losing the carry bit when the shift is performed. If the shift ▶



left produces a carry, the whole sum is exclusively OR'ed with a constant value.

A simple sum check will not distinguish, for example, between 5 4 3 2 and 2 3 4 5 – both will provide the same result. The cyclic redundancy check would provide a different result in this case, making it more reliable.

For the sake of speed, the CRCK algorithms in this program are implemented as machine-code subroutines.

The Protocol

For more details on the Christensen protocol, see *Your Computer*, June 1983. Briefly, however, data is transmitted in 128-byte blocks. Each block starts with an SOH (hex 01), followed by the block number, followed by the 1s complement of the block number (for integrity checking). Then follow exactly 128 bytes of data – all eight bits are transmitted, so object files or data of any kind can be transmitted. Then, there is either a single byte simple sum check, or two bytes of cyclic redundancy check data. The receiving end sends an ACK (hex 06) if it received the block correctly, or a NAK (hex 15) if it didn't. After the last block, the sender transmits an EOT (hex 04) to indicate end of transmission.

Files are transferred at a rate of about 1K per 45 seconds.

What The Program Will Do

The program has the following capabilities:

Full-duplex terminal
Half-duplex terminal
Transmit a file
Receive a file
Analyse a file
Type the last file
Cancel a transmission

These are explained below:

'Full-duplex terminal' is the default mode when the program first commences. It is the correct mode for conversing with a remote bulletin board – such as the MiCC bulletin board. Since Commodore 64s use a non-standard code set (not ASCII), the program automatically converts data typed at the keyboard to standard ASCII. This basically involves reversing upper/lower case, and changing certain control codes (such as backspace, clear screen) to standard ASCII. The only control codes supported are RETURN, clear screen (press SHIFT and CLR/HOME), backspace (press INST/DEL), and the left/right arrow key. To leave terminal mode, press the 'Commodore logo' key.

The 'half-duplex terminal' mode should be used if you are conversing with another Commodore 64 owner. In this case, what you type appears on the screen in light blue; what the other person types appears on the screen in white.

'Transmit' a file initiates transmission of a file to the other end of the line. Before transmitting you should ensure that the other end is about to enter 'Receive' mode (within 60 seconds) or you will get a timeout and the transmission will be aborted. After selecting 'transmit', you will be asked if the file is on disk or cassette, and what its name is. The file will then be loaded, an estimated transmission time (and the number of blocks in the file) will be displayed, and transmission will commence. An asterisk will be displayed as each block is transmitted. Any transmission errors will be displayed in red. If the words 'File transmitted successfully' appear, the file was transmitted correctly. Once the file has been transmitted, the program automatically re-enters terminal mode so you can talk to the other end again.

'Receive a file' initiates reception of a file from the other end of the line. You should ensure the other end is about to transmit a file before entering this mode. In the case of remote CP/M systems (such as the MiCC bulletin board), you should call up XYAM and command it to send the file you want like this:

XYAM S filename

As soon as you have done that, press the Commodore key (to return to the Main Menu) and enter 'R' (for Receive).

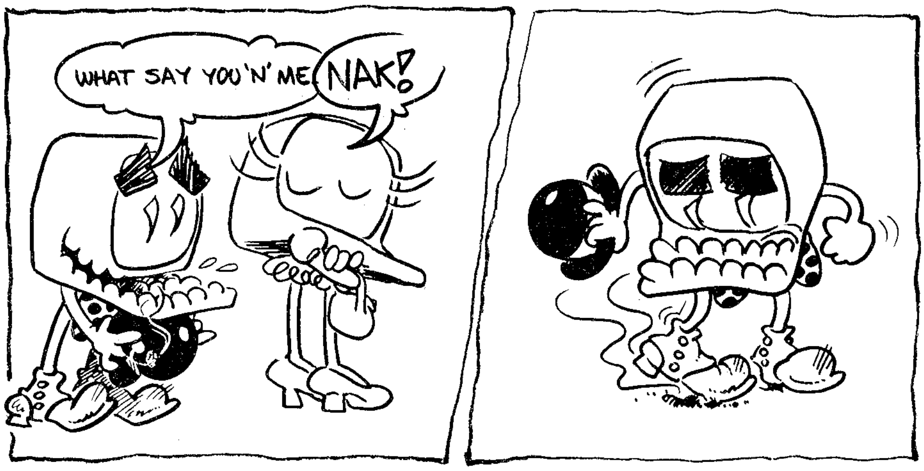
Following reception of a file, the program displays a 'file cyclic redundancy check'. This should agree with the value displayed at the sender's end prior to transmission (or, if the other end is using YAM they should type: CRCK filename). If these figures agree, you can be pretty certain that the file was received correctly.

Once the file has been successfully received, you will be asked whether to save it to disk or cassette and to enter its file name. When the file is saved, the program automatically verifies it to make sure that it saved correctly. At the end of this procedure, the program automatically re-enters terminal mode and you can talk to the other end again.

'Analyse a file' loads a specified file into memory and displays its file size (number of transmission blocks), memory size (in K), file cyclic redundancy check, and the estimated transmission time.

'Type last file' types on the screen the last file that was sent, received or analysed. (So, to display the contents of any file, just Analyse and Type it). Press the SHIFT key to temporarily halt the display, and the Commodore logo key to abort the display and return to the Main Menu. Files which are 'tokenised' or not stored as straight ASCII text files (such as BASIC or G-Pascal files) may display a little strangely.

'Cancel a transmission' cancels a transmission that you commenced in error. First, abort the transmit or receive function by pressing RUN/STOP, then re-run the program and select the 'cancel' function. This will transmit three CAN (hex 18) characters to the other end which should cause the program to abort its transmission/reception.



Colours

The program uses colour coding to identify the different messages and generally avoid confusion. The codes are as follows:

Grey and green – messages (not errors) from the program.

Red – error messages from the program.

Light-blue – data typed by the user at this end.

White – data sent from the other end.

Limitations

The program cannot handle files greater than 24K in length, as it has to load the whole file into memory at once. Files larger than this will corrupt the G-Pascal compiler.

The program can only handle 'program'-type files (that is, files of type 'prg' on disk). This includes BASIC, G-Pascal ▶

and machine-code files in general. With a bit of work you could change from loading files to opening them and reading a byte at a time. This would remove both these restrictions.

The program will not transfer in 'batch' mode (multiple files at one time), unlike YAM.

Future Enhancements

The program could have further features added, but what is presented here is certainly adequate for transferring files backwards and forwards. Once you have this version operational, you can always download improved versions from bulletin boards as they are made available.

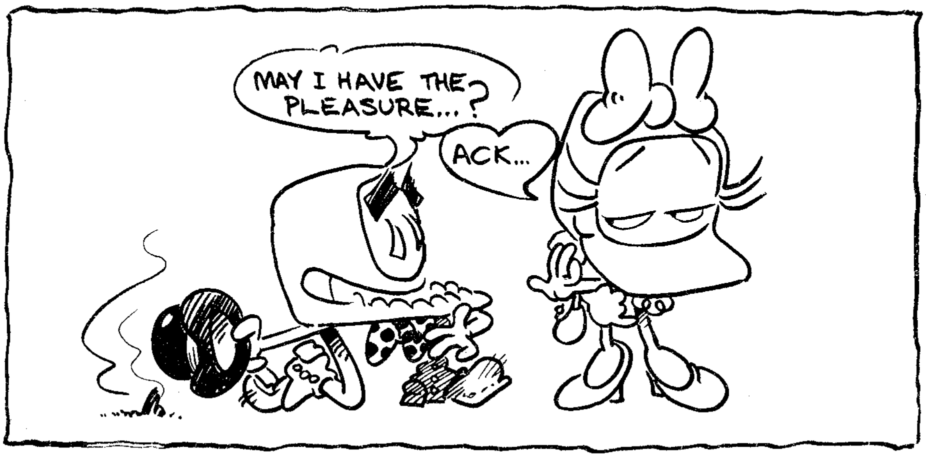
Possible enhancements would be:

1. Implement a 'batch' mode compatible with YAM.
2. Transfer all file types (not just programs) by opening a disk file and reading a byte at a time.
3. Save conversations in memory for later review, with an option to dump a conversation to disk.

Public Domain

Readers are encouraged to give away copies of this program to friends, as we would like to promote the use of the Christensen protocol for data transmission. Do not give away the G-Pascal compiler however, as that is a commercial product and subject to copyright.

If you want to save the effort of typing in the program, copies on disk may be obtained by sending $20 (for postage and duplication costs) to: Gambit Games, P.O. Box 124, Ivanhoe 3079. Computer clubs are encouraged to obtain a copy and make further copies available to members. An Apple version of the program is also available, at the same price, from the same address. ❑



```
(* YAM-compatible modem communication program

   written in G-Pascal for the Commodore 64

   Author: Nick Gammon.    Public Domain Program.

   %a $840  (P-codes start at $840)
*)

const
        bs = 8;
        ff = 12;
        cr = 13;
        fs = 28;
        ctrlz = $1a;
        home = 147;
        true = 1;
        false = 0;

        display_file = false;
        receive_with_crck = true;
        max_retries = 6;
        charcolour = 10;
        white = 1;
        green = 5;
        light_red = 10;
        light_green = 13;
        light_blue = 14;
        light_grey = 15;

        start_address = $1e00;
        cassette = 1;
        disk = 8;
        areg = $2b2;
        xreg = $2b3;
        yreg = $2b4;
        cc = $2b1;
        setlfs = $ffba;
        setnam = $ffbd;

        soh = $1;
        eot = $4;
        ack = $6;
        nak = $15;
        can = $18;
        rs232_status = $297;
        empty = 8;

var
    command : char ;

    buffer : array [130] of char ;
    name1, name2 : array [20] of char ;
    last_terminal_mode,
    medium,
    got_medium,
    length,
    bad_result,
    next_address,
    final_address,
    retries,
    eof,
    abort,
    bad_block,
    seq_error,
    bad_sum_check,
    timeout,
    block_no,
    inverse_block_no,
    expected_block,
    last_block,
    want_crck,
    sum_check_received,
    sum_check_received_2,
    sum_check,
    sum_check_2  : integer ;
    routine : array [35] of integer ;

function commodore_logo;
(***********************)
begin
  commodore_logo := memc [653] and 2 <> 0
end ;

function shift_key_pressed;
(**************************)
begin
  shift_key_pressed := memc [653] and 1 <> 0
end ;

procedure open_rs232_file;
(*************************)
const
      openit = $ffc0;
var name : array [1] of char ;
begin
(* first set up the file name
   as per the RS232 paramters *)

  name [1] := 6;  (*  300 baud  *)
  name [0] := 0;    (*  3-line    *)
  memc [$f8] := $c1; (* buffer *)
  memc [$fa] := $c2; (* buffer *)
  memc [areg] := 2;
  memc [xreg] := 2; (* RS232 *)
  memc [yreg] := 2;
  call (setlfs);
  memc [areg] := 2;
  memc [xreg] := address (name[1]);
  memc [yreg] := address (name[1]) shr 8;
  call (setnam);
  call (openit)
end ;

procedure init;
(*************)
const colour = 1;
      point = 2;
      behindbk = 6;

var i : integer ;

procedure insert(x, y, z);
begin
  routine [i] := x;
  routine [i - 1] := y;
  routine [i - 2] := z;
  i := i - 3
end ;

begin (* init *)
  write (chr (home));
  graphics (charcolour, light_grey);
  memc [650] := 128;  (* all keys auto-repeat *)
  writeln ("YAM-compatible Modem Program for C64.");
  writeln ("Written by Nick Gammon in G-Pascal.");
  writeln ("Version 1.2  - PUBLIC DOMAIN.");
  writeln ("G-Pascal is produced by Gambit Games -");
  writeln (" enquiries: Gambit Games, P.O. Box 124,");
  writeln (" Ivanhoe, Victoria 3079. Australia.");
  writeln ;
  i := 35;
  (* crck routine for transmission *)
  insert($8500a9,$5f855e,$854bb1);
  insert($08a207,$260726,$5f265e);
  insert($a50c90,$10495f,$a55f85);
  insert($21495e,$ca5e85,$88e9d0);
  insert($60e0d0,0,0);
  (* crck routine for file *)
  insert($8500a9,$068505,$0506a8);
  insert($080626,$184bb1,$850565);
  insert($902805,$97490a,$a50585);
  insert($a04906,$e60685,$02d04b);
  insert($a54ce6,$5ec54b,$a5dbd0);
  insert($5fc54c,$a5d5d0,$4b8505);
  insert($8506a5,$ff604c,0);
  buffer [128] := 0;
  buffer [129] := 0;
  command := "f";
  definesprite (32,
   $ff,$ff,$ff,$ff,$ff,$ff,$ff,$ff);
  sprite (1, point, 32,
          1, colour, light_grey,
          1, behindbk, true);
  got_medium := false;
  final_address := start_address;
  open_rs232_file
end ; (* of init *)

procedure start_error;
(********************)
begin
  graphics (charcolour, light_red);
  writeln
end ;

procedure error;
(**************)
begin
if expected_block <> -1 then
   write (" on block ",
              expected_block)
else
   write (" on EOT");
writeln (" retry ", retries);
retries := retries + 1;
graphics (charcolour, green);
if retries > max_retries then
  abort := true
end ;

procedure get_file_name;
(**********************)
var i, got_cr : integer ;
     ch : char ;
begin
if not got_medium then
  begin
  writeln ;
  write ("<D>isk or <C>assette? ");
  graphics (charcolour, light_blue);
  repeat
    read (ch);
    ch := ch and $7f
  until (ch = "d")
     or (ch = "c");
  writeln (chr (ch));
  graphics (charcolour, green);
  if ch = "d" then
    begin
    medium := disk;
    open (15, disk, 15, "i")
    end
  else
    medium := cassette;
  got_medium := true
  end ;
repeat
  writeln ;
```

▶

Continued from page 72

```
  write ("File name? ");
  graphics (charcolour, light_blue);
  read (name1);
  graphics (charcolour, green);
  got_cr := false;
  for i := 0 to 20 do
    if not got_cr then
      begin
      name2 [20 - i] := name1 [i];
      if name1 [i] = cr then
        begin
        length := i;
        got_cr := true
        end
      end
until length <> 0
end ;

procedure check_result;
(*********************)
const readst = $ffb7;

var i, error_code : integer ;
    result : array [80] of char ;
begin
  if memc [cc] and 1 then
    error_code := memc [areg]  (* got error *)
  else
    begin
    call (readst);
    error_code := memc [areg] and $bf
    end ;
  bad_result := error_code;
  if medium = disk then
    begin
    get (15);
    read (result);
    get (0);
    result [80] := cr;
    if (result [0] <> "0")
    or (result [1] <> "0") then
      begin
      bad_result := true;
      i := -1;
      start_error;
      repeat
        i := i + 1;
        write (chr (result [i]))
      until result [i] = cr
      end
    end ;
  writeln ;
  if error_code then
    begin
    start_error;
    writeln ("File error, code: ",
                error_code)
    end ;
  graphics (charcolour, green);
  if not bad_result then
    writeln ("Ok.")
end ;

procedure load_nominated_file (flag);
(***********************************)

procedure load_file;
(******************)
const
      loadit = $ffd5;

begin
  memc [areg] := 1;
  memc [xreg] := medium;
  memc [yreg] := 0; (* relocate *)
  call (setlfs);
  memc [areg] := length;
  memc [xreg] := address (name2[20]);
  memc [yreg] := address (name2[20]) shr 8;
  call (setnam);
  memc [areg] := flag;  (* load /verify *)
  memc [xreg] := start_address;
  memc [yreg] := start_address shr 8;
  call (loadit);
  check_result
end ;

(***** start of : load_nominated_file ***)
begin
repeat
  if flag = 0 then  (* load *)
    get_file_name;
  load_file
until (bad_result = 0)
   or (flag = 1)
end ;

procedure save_nominated_file;
(****************************)

procedure save_file;
(*******************)
const saveit = $ffd8;
      register = $6a;
begin
  memc [areg] := 1;  (* file no *)
  memc [xreg] := medium;
  memc [yreg] := 0;
  call (setlfs);
  memc [areg] := length;
  memc [xreg] :=
         address (name2 [20]);
  memc [yreg] :=
         address (name2 [20]) shr 8;
  call (setnam);
  memc [register] := start_address;
  memc [register + 1] :=
                start_address shr 8;
  memc [areg] := register;
  memc [xreg] := final_address;
  memc [yreg] := final_address shr 8;
  call (saveit);
  check_result
end ;

(***** start of : save_nominated_file ***)
begin
repeat
  get_file_name;
  save_file;
  if not bad_result then
    begin
    if medium = cassette then
      begin
      writeln ;
      writeln ("Rewind cassette to save point for");
      writeln ("verification - press <SHIFT> when ready.");
      repeat until shift_key_pressed
      end ;
    load_nominated_file (1)   (* verify save *)
    end
until not bad_result
end ;

function from_modem;
(******************)
begin
  get (2);
  from_modem := getkey ;
  get (0)
end ;

procedure display_char (x);
(*************************)
begin
  x := x and $7f;

(* Reverse upper/lower case *)

  if (x >= $61) and
     (x <= $7a) then
     x := x - $20
  else
  if (x >= "a") and
     (x <= "z") then
     x := x + $20;

(* Only display if printable *)

  if (x >= " ")
  or (x = cr) then
     write (chr (x))
  else
    if x = bs then
      write (chr (157))
    else
      if x = fs then
        write (chr (29))
      else
        if x = ff then
          write (chr (home))
end ;

procedure to_modem (x);
(*******************)
begin
  put (2);
  write (chr (x));
  put (0)
end ;

function calc_crck;
(****************)
begin
  memc [$4b] := address (buffer [130]);
  memc [$4c] := address (buffer [130]) shr 8;
  memc [yreg] := 130;
  call (address (routine[35]));
  calc_crck := mem [$5e] and $ffff
end ;

procedure calc_file_crck;
(**********************)
begin
  memc [$4b] := start_address;
  memc [$4c] := start_address shr 8;
  memc [$5e] := final_address;
  memc [$5f] := final_address shr 8;
  call (address (routine[20]));
  writeln ("Cyclic redundancy check = $",
              hex (mem [$4b] and $ffff));
end ;

function next_char (period);
(**************************)
const count_per_second = 145;
var ch : char ;
    counter : integer ;
begin
counter := period * count_per_second;
repeat
  ch := from_modem;
  counter := counter - 1
until (not (memc [rs232_status] and empty))
   or (counter <= 0);
timeout := memc [rs232_status] and empty <> 0;
next_char := ch
end ;

procedure purge;
(*************)
var discard : char ;
begin
repeat
  discard := next_char (1)
until timeout
end ;

procedure send_nak;
(****************)
begin
  purge;
  if (expected_block = 1)
  and want_crck then
    to_modem ("c")
  else
    to_modem (nak)
end ;

procedure cancel_trans;
(*************)
begin
  purge;
  to_modem (can);
  to_modem (can);
  to_modem (can);
  start_error;
  writeln ("Transmission aborted")
end ;

procedure receive_block;
(*********************)
var ch : char ;
    i : integer ;
begin
bad_block := false;
block_no := next_char (1);
if not timeout then
  inverse_block_no := next_char (1);
if (block_no + inverse_block_no + 1)
   and $ff <> 0 then
  begin
  start_error;
  write ("Bad block no.");
  error;
  send_nak;
  bad_block := true
  end
else
  if ((block_no = last_block and $ff)
  and (expected_block <> 1))
  or (block_no = expected_block and $ff) then
    seq_error := false
  else
    begin
    seq_error := true;
    start_error;
    writeln ("Block number sequence error")
    end ;
if not (bad_block or seq_error) then
  begin
  sum_check := 0;
  for i := 0 to 127 do
    if not timeout then
      begin
      ch := next_char (1);
      buffer [i] := ch;
      sum_check := sum_check + ch
      end ;
  if not timeout then
    sum_check_received := next_char (1);
  if want_crck then
  if not timeout then
    sum_check_received_2 := next_char (1);
  if timeout then
  begin
    start_error;
    write ("Timeout on receive");
    error;
    send_nak
    end
  else
    begin
    bad_sum_check := true;
    if want_crck then
      if calc_crck = sum_check_received shl 8
      or sum_check_received_2 then
        bad_sum_check := false
      else
    else
      if sum_check and $ff =
         sum_check_received then
        bad_sum_check := false;
    if bad_sum_check then
      begin
      start_error;
      write ("Sum check error");
      error;
      send_nak
      end
    else
      begin
      to_modem (ack);
      retries := 0;
      if block_no = expected_block and $ff then
        begin
        last_block := expected_block;
        expected_block := expected_block + 1;
        if display_file then
          for i := 0 to 127 do
            display_char (buffer [i])
        else
          write ("*");
        for i := 0 to 127 do
          begin
          memc [next_address] :=
               buffer [i];
          next_address := next_address + 1
          end
        end
      end
    end
  end
end ;

procedure receive_block_can_eot;
(*****************************)
var ch : char ;
begin
repeat
  ch := next_char (10)
until (ch = soh)
   or (ch = eot)
   or (ch = can)
   or timeout;
if timeout then
  begin
  start_error;
  write ("Timeout at start");
```

```
    error;
    send_nak
    end
else
  case ch of
    soh: receive_block;
    can: begin
         start_error;
         writeln ("Sender CANcelled transmission");
         abort := true
         end ;
    eot: begin
         eof := true;
         to_modem (ack)
         end
  end   (* of case *)
end ;

procedure receive_file;
(*********************)
begin
writeln ;
graphics (charcolour, light_green);
writeln ("----- Receive a File -----");
graphics (charcolour, green);
writeln ;
expected_block := 1;
last_block := 0;
retries := 0;
abort := false;
eof := false;
seq_error := false;
next_address := start_address;
want_crck := receive_with_crck;
send_nak;  (* get things going *)
repeat
  receive_block_can_eot
until abort or eof or seq_error;
writeln ;
if eof then
  begin
  final_address := next_address;
  writeln ;
  writeln ("File received successfully");
  calc_file_crck;
  save_nominated_file
  end
else
  begin
  final_address := start_address;
  cancel_trans  (* stop other end *)
  end
end ;

procedure analyse_file;
(*********************)
var
    file_length, blocks, mins : integer ;
begin
writeln ;
load_nominated_file (0);
final_address := memc [xreg] + memc [yreg] shl 8;
file_length := final_address - start_address;
while file_length and $7f <> 0 do
  begin
  file_length := file_length + 1;
  memc [final_address] := ctrlz;
  final_address := final_address + 1
  end ;
blocks := (final_address - start_address)
           / 128;
mins := blocks * 561 / 600;
writeln (blocks, " blocks, ",
          blocks * 10 / 80,
          ".",
          blocks * 10 / 8 mod 10,
          " K");
calc_file_crck;
writeln ("Transmission time: ",
          mins / 10, ".",
          mins mod 10,
          " minutes.")
end ;

procedure process_can;
(********************)
begin
  start_error;
  writeln ("Receiver CANcelled transmission");
  graphics (charcolour, white);
  abort := true
end ;

procedure transmit_block;
(***********************)
var ch : char ;
    discard,
    i : integer ;

procedure get_ack;
(****************)
begin
  ch := next_char (10);  (* wait for ack *)
  if timeout then
    begin
    start_error;
    write ("Timeout on ACK");
    error
    end
  else
  if ch = can then
    process_can
  else
  if ch <> ack then
    begin
    start_error;
    write ("Got ",ch," for ACK");
    error
    end
end ;  (* of get_ack *)

begin
sum_check := 0;
for i := 0 to 127 do
  begin
  ch := memc [next_address];
  next_address := next_address + 1;
  sum_check := sum_check + ch;
  buffer [i] := ch
  end ;
if display_file then
for i := 0 to 127 do
  display_char (buffer [i])
else
  write ("*");
if want_crck then
  begin
  sum_check_2 := calc_crck;
  sum_check := sum_check_2 shr 8;
  sum_check_2 := sum_check_2 and $ff
  end ;
retries := 0;
inverse_block_no := block_no xor $ff;
expected_block := block_no;
repeat
  to_modem (soh);  (* start block *)
  to_modem (block_no);
  to_modem (inverse_block_no);
  for i := 0 to 127 do
    begin
    discard := from_modem;  (* ignore any spurious glitches *)
    to_modem (buffer[i])
    end ;
  to_modem (sum_check);
  if want_crck then
    to_modem (sum_check_2);
  get_ack
until abort or ((not timeout) and (ch = ack));
if next_address >= final_address then
if not abort then
  begin
  retries := 0;
  expected_block := -1;
  repeat
    to_modem (eot);
    get_ack
  until abort or ((not timeout) and (ch = ack));
  if not abort then
    eof := true
  end ;
block_no := block_no + 1
end ;

procedure send_file;
(******************)
var ch : char ;
begin
writeln ;
graphics (charcolour, light_green);
writeln ("----- Send a File -----");
graphics (charcolour, green);
analyse_file;
next_address := start_address;
block_no := 1;
expected_block := 1;
abort := false;
eof := false;
retries := 0;
purge;  (* empty buffer *)
writeln ; writeln ;
writeln ("Awaiting initial NAK");
repeat
  ch := next_char (60);  (* wait a minute *)
  if timeout then
    begin
    start_error;
    writeln ("No response from other end");
    end
  else
    begin
    if ch = nak then
      want_crck := false
    else
      if ch = "c" then
        want_crck := true
      else
        if ch = can then
          process_can
        else
          begin
          start_error;
          write ("Got ",ch," for NAK");
          error
          end
    end
until (ch = nak) or (ch = "c")
   or timeout or abort;
if not (timeout or abort) then
  repeat
    transmit_block
  until abort or eof;
if eof then
  begin
  writeln ;
  writeln ("File transmitted successfully")
  end
else
  cancel_trans  (* stop other end *)
end ;

procedure terminal_mode (half_duplex);
(************************************)
const active = 7;
var input : char ;
    x : integer ;
begin
last_terminal_mode := command;
graphics (charcolour, green);
writeln ;
graphics (charcolour, light_green);
write ("Terminal Mode - ");
if half_duplex then
  write ("Half")
else
  write ("Full");
writeln (" duplex");
writeln ("Press <Commodore> key for Main Menu");
writeln ;
graphics (charcolour, white);
sprite (1, active, true);
repeat
  x := cursorx ;
  if x > 40 then
    x := x - 40;
  positionsprite (1,
    x * 8,
    cursory * 8 + 42);
  input := from_modem;
  if input <> 0 then
    display_char (input);
  input := getkey ;
  if input <> 0 then
    begin
    if (input >= $c1) and
       (input <= $da) then
      input := input - $60;
    if input = $8d then
      input := cr
    else
      if (input = $9d)
      or (input = $14) then
        input := bs
      else
        if input = 29 then
          input := fs
        else
          if input = home then
            input := ff;

(* Reverse upper/lower case *)

    if (input >= $61) and
       (input <= $7a) then
       input := input - $20
    else
    if (input >= "a") and
       (input <= "z") then
       input := input + $20;
    to_modem (input);
    if half_duplex then
      begin
      graphics (charcolour, light_blue);
      display_char (input);
      graphics (charcolour, white)
      end
    end
until commodore_logo;
sprite (1, active, false)
end ;

procedure type_file;
(******************)
begin
next_address := start_address;
writeln ;
writeln ("Press <Commodore> key to abort list");
writeln ("      <SHIFT>     key to pause list");
writeln ;
graphics (charcolour, light_green);
while (next_address < final_address)
  and not commodore_logo do
  begin
  repeat
  until not shift_key_pressed;
  display_char (memc [next_address]);
  next_address := next_address + 1
  end ;
writeln
end ;

(* -------- MAIN PROGRAM -------- *)
begin
  init;  (* ready for crck *)
  repeat
    graphics (charcolour, green);
    case command of
     "a": analyse_file;
     "c": cancel_trans;
     "f": terminal_mode (false);
     "h": terminal_mode (true);
     "r": receive_file;
     "s": send_file;
     "t": type_file
    end ;  (* of case *)
    if (command = "s")
    or (command = "r") then
      command := last_terminal_mode
    else
      begin
      graphics (charcolour, green);
      writeln (chr (14));  (* lower case *)
      writeln ("<A>nalyse a file");
      writeln ("<C>ancel transmission");
      writeln ("<F>ull duplex terminal");
      writeln ("<H>alf duplex terminal");
      writeln ("<R>eceive a file");
      writeln ("<S>end a file");
      writeln ("<T>ype last file");
      writeln ("<Q>uit program");
      writeln ;
      write ("Command? < >",chr (157),chr (157));
      graphics (charcolour, light_blue);
      repeat
        read (command);
        command := command and $7f
      until (command = "f")
         or (command = "s")
         or (command = "q")
         or (command = "a")
         or (command = "c")
         or (command = "h")
         or (command = "r")
         or (command = "t");
      writeln (chr (command))
      end
  until command = "q";
  close (2)
end .
```