This document was produced on the Apple Macintosh, using Microsoft Word 3.01 for text, and MacDraw II for diagrams
Printed on the Apple Laserwriter IINTX
Set in 10-point ITC Bookman

By: Ben Hekster, Tankelanden 5, 7542 DR Enschede, Netherlands. 053-764091

Contents

Preface

This is the report written towards the requirements of the '100-Hour Assignment' (Dutch: *100-Uurs Opdracht*) course for the degree of Electrical Engineering at the University of Twente in the Netherlands. The assignment was executed in the Measurement Technology Laboratory (*Mittaustekniikan Osasto*) at the University of Oulu in Finland, under Prof. Saarimaa and with the guidance of Prof. Kalliomäki.

The report consists of two main sections—the first of these, entitled "Development", is in the form and contains the type of information typically expected in a report of this kind—i.e., definition of the problem which the work described in this report attempts to solve, my solution of the problem, and consideration of the various alternatives which were considered but abandoned during the design process. Also, some ideas are proposed for further improvements which might be implemented in the future.

The second section in the report, "Reference", takes the form of a reference manual, describing in detail the information needed to use the software which was produced in the solution of the problem—the 'ANSWER' program. The software has been carefully designed so as to be easily applicable in a wide variety of applications, but sufficient detail is given for programmers who may wish to make their own adjustments or additions to the program.

Therefore, if you are only reading this report in order to find out how to use ANSWER, or if you are looking for details on the program's operation, you may wish to proceed directly to the Reference section. However, if you are completely unfamiliar with the program, the last section in the "Remote Communication" chapter, "Implementation & Testing", may prove illuminating. If you are interested in the history and development of the program, continue reading from this point.

The entire project—starting from its definition through to the writing of the original report—was concluded with great exertion in two and a half weeks' time.

—Ben Hekster

Introduction

In this chapter, I will discuss the reasons behind the commissioning of my work. First, however, I will begin by elaborating slightly on the background of the environment in which it is to be applied.

In maintaining the official Finnish time and frequency standards by the Finnish State Research Institute (*Valtion Teknillinen Tutkimuskeskus,* or VTT), certain measurements need to be made relating to the variation of the transmission delay of a signal received at specific locations throughout Scandinavia. These measurements are made at each of the locations, and are later collected at VTT and statistically correlated for seasonal, circadian, or other periodic variations. (The exact nature or purpose of these calculations is unknown to me, but has absolutely no bearing on this work.)

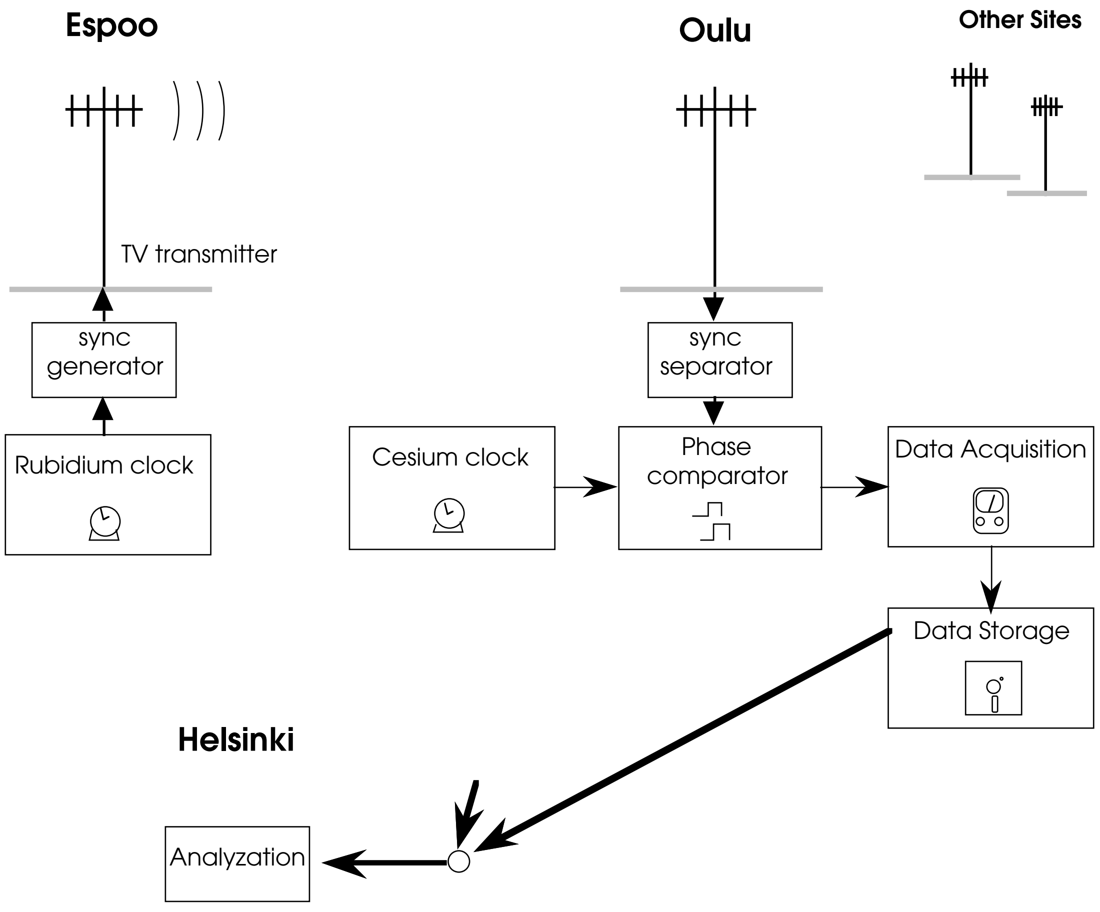


Fig. 1. The measurement scheme

The measurements taken are samples of phase differences between a locally generated Cesium-based oscillator clock signal and a received signal related to the common reference clock in Espoo (see Fig. 1). This reference signal is derived from the vertical retrace signal of national television station YLE 1, which in turn is driven by a relatively stable Rubidium-oscillator clock located at VTT. As the instability of this reference Rubidium clock is tracked and recorded, against a number of other clocks at VTT, the phase difference measurements can later be used with these recordings to determine the variation of the signal transmission delay.

Several phase measurements are taken at 10-minute intervals by an IBM PC-compatible computer from different channels of an IEEE-488 bus, and are temporarily stored in a buffer in memory. From this buffer, the data is appended to two different files—one receives data from selected channels at the full 10-minute temporal resolution (called *'quick' data*) while the other file gets only two-hour averages, but from all channels (*'super' data*). New files are used every 10 days. By using a formula to convert Julian dates to an integral number of days since a certain celestial event, and using this number in the file name, future file indexing becomes straightforward.

As the situation existed prior to my work, the phase difference measurement data from the different institutions were collected on 360K floppy disks which were mailed to Helsinki. On some occasions the data on a disk had become corrupted, so that another one needed to be mailed—altogether a rather slow, cumbersome and unnecessarily expensive procedure. I therefore proposed to design a system which would employ electronic file transmission to supplant the movement of the data on physical media. It was also desirable for each of the installations to have access to the collective body of measurement data present in Helsinki, something which up to then was only possible by phoning an individual at VTT with physical access to the data, and asking him to mail a disk back in return. In this report, I will refer to these issues collectively as the **remote communication** problem. The problem I attempted to solve is actually quite independent of the particular nature of the data, i.e., the way in which it happens to have been generated, or what it represents.

More or less unrelated to this (the communication of data between *different* installations), there was an additional desire at the Measurement Technology Laboratory to be able to communicate data between personal computers throughout the laboratory electronically rather than physically. The nature of this problem, which I will hereafter refer to as the **local communication** problem, is very similar to that of the remote communication, but as we will shortly see, the completely different scale of ranges required a somewhat different solution.

I will refer to the entire scheme, comprising hardware, software, protocol, and such, as simply the **system**. I have already used the term **installation** to describe different physical locations in the remote communication problem. A single instance of a communications session is called a **transaction**—this not only to stress that the communications protocols require both parties to actively negotiate in establishing the connection,† but also to indicate that either party may wish to receive data from the other, i.e., two-way communication.

† Just refer to any of the levels in the ISO/OSI Reference Model [4]. In practical terms, an answering modem may adapt itself to the baud rate of an originating modem. Or, a file transfer program may downgrade its own level of error checking if the other party cannot support it. In fact, both of these examples highlight instances of negotiation occurring in this very system

Remote Communication

After having defined the problems, I then proceeded to weigh various alternative solutions. I will start by discussing my solution to the remote communication problem in this chapter, following in the next with local communication. This dichotomy does not reflect the actual development of the systems, which occurred more or less concurrently, but is conducive to a good understanding of the underlying concepts and decisions made in the process.

Design

Although it has not been mentioned explicitly yet, it seems almost immediately obvious that phone lines should be used for the remote communication. Although there are other alternatives, such as the usage of packet switching-networks, there were no such facilities available at the Measurement Technology Laboratory. Note that using phone lines implies that each installation have a modem installed in one of its computers.

Whichever medium is selected for the actual data transfer, several basic alternative communications methods may be considered. Each of these has its own distinctive impact on system characteristics, such as the amount of human activity required in effecting the transaction, intelligence of the system, security, and ease of implementation (complexity):

- *Nonautonomous transaction* The data is transmitted electronically, but requires human attendance of the system on both sides of the communication. This is by far the easiest to implement, since the system can rely on human intelligence to detect and resolve any contingencies which may arise while establishing or during the transactions, and which it has no knowledge of. The system therefore needs very little intelligence of its own. Security is not an issue, since it has already implicitly been resolved when the transaction is initiated—presumably by both operators vocally identifying themselves to each other over the phone, before the transmission is started.
- *Semiautonomous transaction* The transactions only require human action on one side of the communication. Significant intelligence is already required on the autonomous side—security starts to become an issue, since there are now no humans to identify the party initiating the transaction. Records of all transactions must be kept at the autonomous end of the communication. The problem of scheduling the transactions between the various installations is now delegated to the humans *initiating* the transactions. If the system is to be effective, it must be reliable enough to prevent the operators becoming frustrated with it and reverting to manual operation.
- *Autonomous transaction* The system is fully automatic—data is transmitted completely transparently to humans at all installations. At regular intervals, the system decides which of the data to transmit, and initiates the communication itself. A highly intelligent system is required to initiate a coordinated sequence of transactions, to cope with contingencies such as may occur, for instance, if an installation being contacted is unavailable. In such cases, records must be kept of which installations have received which data, and subsequent attempts must be initiated to transfer data which has not yet been received by systems which may, for example, have been temporarily out of commission. The remarks relating to security and reliability in the semiautonomous transfer approach apply even more strongly here.

After deliberation with professors at Oulu and officials at VTT in Helsinki, I opted for the second alternative, the semiautonomous communication. The situation is depicted in the following figure:

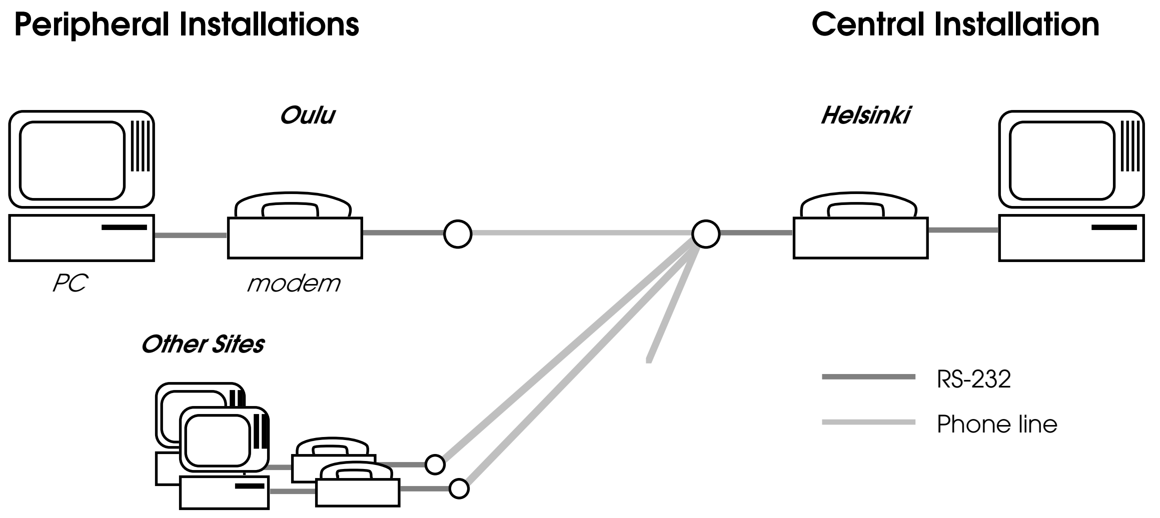


Fig. 2. Schematic representation of the remote communication solution

The autonomous party in the communication is aptly named the **central installation**, while the remaining installations are called **peripheral installations**. The terms 'central' and 'peripheral' are used as adjectives generally. The designation of VTT in Helsinki as the central installation is natural, as all the communication *is* centered on this installation. As the central installation is distinct from all others, transferring files in this direction may be referred to as **uploading**, while the converse is called **downloading**.

The semiautonomous is preferred to the nonautonomous method, since it relieves the operators in Helsinki of the tedious task of collecting data from various sources, not to mention having to verify the data in each of the files for errors—this task can be handled by the transaction system. Also, operators at the peripheral installations are still able to select a time convenient to them to execute the transaction, rather than having to wait or make an appointment for an operator at the central installation to be present. Better still, peripheral operators may at any time access the central installation to obtain the most recent files uploaded from other peripheral installations.

A fully autonomous system might seem slightly superior in that it is no longer necessary for peripheral operators to manually initiate a transaction. However, peripheral installations must then either dedicate an entire computer to this task alone, or an individual at each site must be designated to start the program and ensure the data is ready, before the next scheduled transaction. This hardly seems much less effort than in the semiautonomous approach, and considering the significant additional complexity such a system would represent certainly not worth the effort. It seems that the fully autonomous method is not well-suited to a personal computer-based system. Had the system been implemented on mainframes with appropriate scheduling capabilities†, this would most certainly have been otherwise. Finally, consider that the semiautonomous method gives individual peripheral operators much more freedom in deciding when to execute their transaction, and the ability to peruse the file list of the central installation at their leisure.

IMPLEMENTATION & TESTING

The implementation of the system proceeded in three main steps, each of which will now be considered separately.

Installation of the hardware. Since phone lines are used for the long-distance communication, the presence of a modem in each of the installations' designated computers is required. Since there was no computer with a modem in the Measurement Technology Laboratory at that time, a modem was acquired which I then proceeded to install. Also, a cable was prepared to connect the modem's *line* input to a phone outlet.

† E.g. in UNIX, the *at* operating system function

The correct operation of the modem and cable were ascertained using the KERMIT terminal emulation facility to send commands to and receive responses from the modem. Also, some calls were made to a dial-in modem of a university mainframe, and some simple commands were executed remotely on the mainframe through the modem to ensure that it was operating correctly, and was capable of establishing a connection with another system.

Preparation of the peripheral installation software. Although it would have been possible to write special software for the peripheral installations, this would have amounted to reinventing the wheel, since no other tasks are required of this software than can already be handled perfectly well by the excellent communications software which already exists—namely, terminal emulation, and file transfer. The terminal emulation capability is required when interacting with the central installation, (e.g., when selecting a file to be downloaded) and the file transfer option when the actual data is transferred. In the immediately preceding section, the KERMIT program is mentioned—as the name implies, other than being a straightforward terminal emulator it also supports the standard Kermit‡ file-transfer protocol. Since this program was already in general use at VTT in Helsinki, and as inspection of the manual showed the software to be perfectly adequate for the purposes at hand, I decided to use it for the peripheral installations.

Note that other software supporting the Kermit file-transfer protocol also exists for the IBM PC-compatibles, which may be used instead of KERMIT if so preferred. In fact, it is not even required that the peripheral installations have to be based on IBM PC-compatibles—any computer may be used, as long as there is software available for it which supports the Kermit protocol.

Writing the central installation software. This was by far the greatest effort of this work. As far as I know, no software preexisted which might have been readily applied. As has been delineated by the preceding matter in this and the previous chapter, a program is required which will operate without human intervention for prolonged periods of time, and will allow the up- and downloading of files at the request of the peripheral operator. The programming effort which resulted from these requirements is the ANSWER program, so named because it does not initiate transactions itself but monitors a phone line for incoming calls and answers them. (Note that this capability requires an *auto-answer* modem at the central installation.) The operation of ANSWER is described in great detail in the aforementioned second section of this report.

In effect, the central installation runs ANSWER unattended on its designated computer. The peripheral operator may use his preferred communications software (e.g., KERMIT) whenever he likes to gain access to the system by identifying himself with a distinct user name and password, provided by the central installation. After the operator's identity has been verified by ANSWER, the program allows him to upload his own measurement data, or to download files he might want. The security facility of the ANSWER program even makes it possible to limit the level of access extended to certain individuals according to a *privilege level* scheme.

As I would not be available after completion of this work to maintain or upgrade the program, I concentrated on writing it in such a way as to make its operation very easily modifiable. This is exemplified by the usage of so-called *scripts,* which allow the way in which the program responds to commands and inquiries from peripheral operators during a transaction to be subdivided into small steps, and changed simply by editing a text file called a *script file.* In fact, the program is so flexible and completely independent of this particular application, that by using a different script file, ANSWER may be used as a bulletin board system for any particular application which may be desired. A sample script file has already been predefined to allow ANSWER to be used in this particular application. Refer to the "Example" section in the "Script File" chapter for an instructive general idea of what scripts actually look like.

One point of interest is that the ANSWER program actually during its execution calls Kermit to effect the actual file transfer. In this manner ANSWER automatically benefits from all the advantages of the Kermit protocol, such as transmission error checking and

‡ In this report, the word *KERMIT* in full capitals indicates the software program for the IBM PC, while *Kermit* with a single capital refers to the file-transfer protocol (which happens to be supported by KERMIT)

automatic packet retransmission. A significant number of parameters of the program, mostly pertaining to the communication protocol, needed to be set before the file transfer would work—these were collected in an initialization file called 'MSKERMIT.INI' (which is read by Kermit when it is executed) so that future users of KERMIT will benefit from them. Peripheral installations already using communications software would already have a similar file themselves. The Kermit protocol negotiates applicable communications parameters such as transmission packet size before the transfer is started, so that it is not required that the installations use identical settings.

It must be said that due to the lack of available time, the sample script which I have written was intended to serve mainly to demonstrate and illustrate the basic capabilities of the script system. As such it is rather terse, and does not fully utilize the full potential of the ANSWER program. Several improvements will be made by users of the system in Oulu, which will greatly improve the operation and general appearance of the user-transaction interface. Some ideas which I have suggested:

- the incorporation of context-sensitive help;
- further specification and application of privilege levels;
- more detailed and user-friendly screen layout and menu hierarchy;
- implementation of additional functions besides simple up- and downloading.

The framework of the program as is should provide enough flexibility to absorb any additions or alterations which may be found necessary in the future.

Local Communication

Regularly, not insignificant amounts of data needed to be exchanged between certain computers within the Measurement Technology Laboratory†. Of course, the data could be moved physically, on floppy disks (which indeed, it was), but a faster, more modern solution was desired.

DESIGN

This part of the assignment was so straightforward that it hardly merits its own 'Design' section. In any case, although the symmetry of the problem with the remote communication one might again suggest the use of telephone lines this was deemed unnecessary, and for some extremely obvious reasons direct cabling using the RS-232 serial communications ports was selected (see Fig. 3):

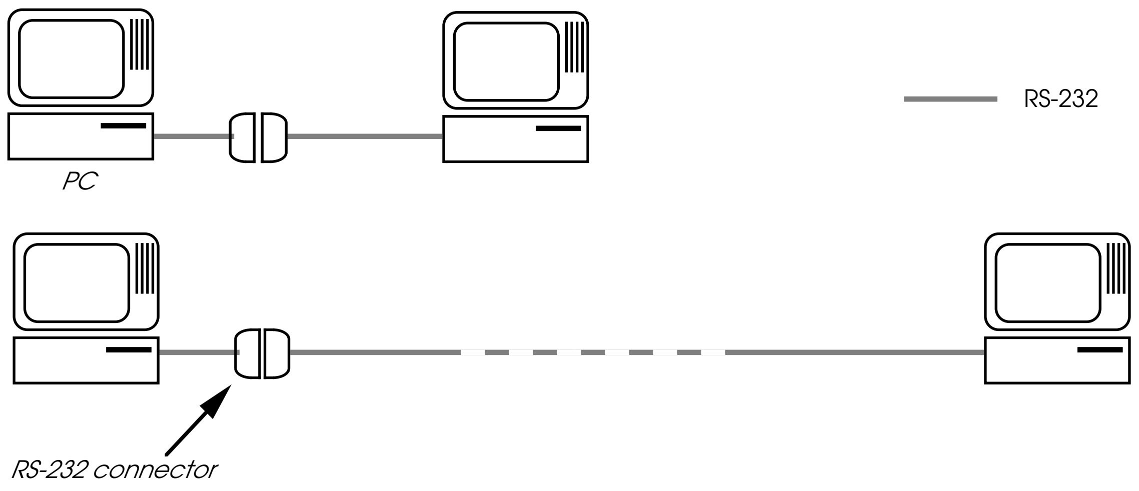


Fig. 3. Schematic representation of the local communication solution

- *Initial cost.* You can't beat the price of two-wire grounded cable. Some experimentation showed that the KERMIT program running on both computers could communicate using only the Transmitted Data (TxD), Received Data (RxD) and Signal Ground (SGND) signals from the RS-232 protocol;
- *Running cost.* Cables, as opposed to phone lines, have no running cost;
- *Speed.* Reasonably-priced modems are usually limited to transmission speeds of up to 2400 baud, while the RS-232 cabling method demonstrated rates of up to 9600 baud (the highest rate the PC-compatible could handle) when it was completed;
- *Reliability.* Phone lines are noisy, connections may be inadvertently broken, or the connection may not be established due to a malfunction in the switchboard. Cables, if they are laid free of obstructions and the connectors are soldered correctly, do not suffer from these problems;
- *Practicality.* Since communication practically always occurred between the same pairs of computers, there was no need for the generality of being able to dial into any other computer. With cabling, then, there is the benefit of not having to dial the number of the other computer on each occasion.

Also, using cabling frees the use of the phone while communicating.

† The nature of this data was not specified further, nor was it necessary that it should be

Implementation & Testing

This is even more straightforward than the previous section. However, two points are worth mentioning. Firstly—since some of the computers between which was to be communicated were situated more than 30 meters apart, there was some concern on my part that the computer's ports would not be able to drive the lines so as to make communication possible, since the EIA RS-232 standard dictates a maximum distance of only 15 meters. After testing, however, several large files (in the order of 200 Kb) were transmitted back and forth (not using a packet retransmission facility) without *any* errors. It goes without saying that the other connections were similarly tested as well.

Secondly—since two computers are always in use during transactions (the central and a peripheral computer), a *null modem cable* was constructed which accurately simulates the presence (on a signal level) of two modems and a telephone line (see Fig. 4). Using this cable, one computer, running ANSWER, represented the central installation, while the other, running KERMIT, a peripheral installation. This proved to be an invaluable aid during testing and evaluation of ANSWER.

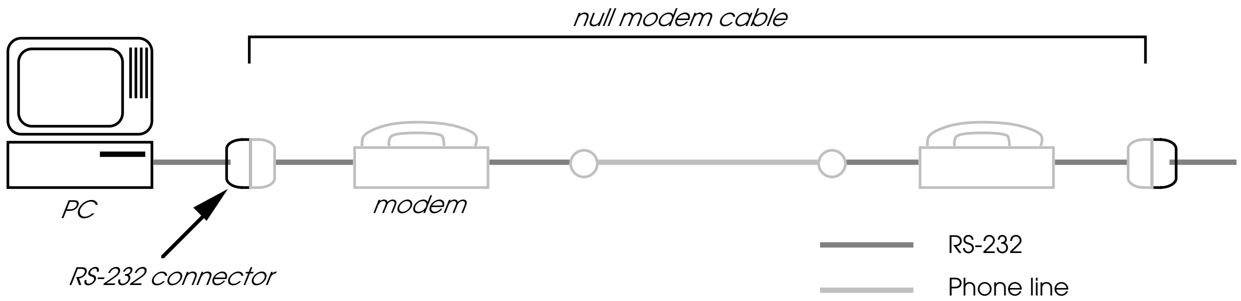
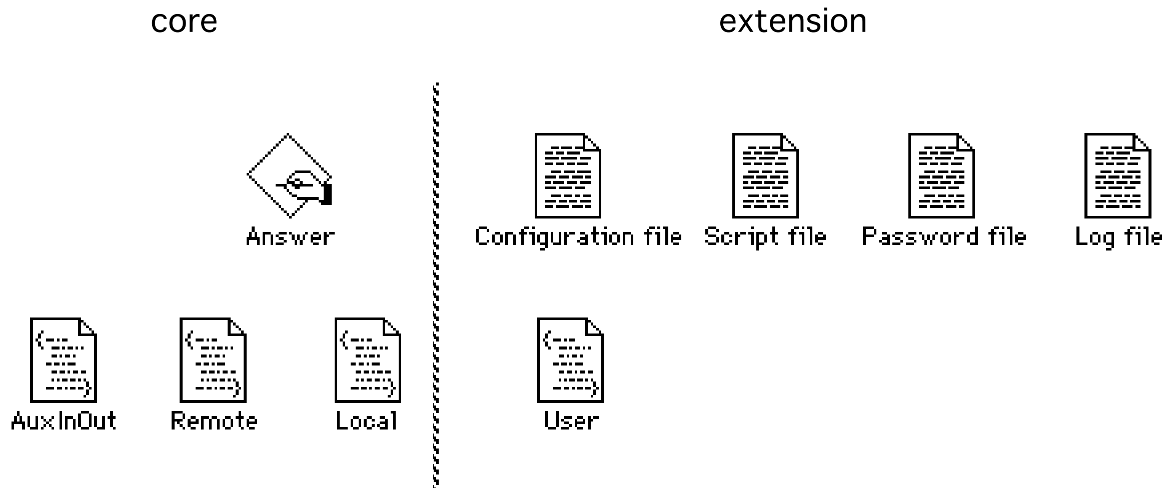


Fig. 4. Schematic representation of the effect of the null modem cable

If electronic communication proves to be very effective, installation of a Local Area Network might be considered.

Answer

The ANSWER program was written in Turbo Pascal Version 3.0, and consists of several *units* and auxiliary files which are used by the program. The various files are shown in the following figure which is followed by a short description of each of them:



Answer unit The main program code is contained in this unit†—it is the one in which execution first begins when the program is started. It reads the *configuration file,* initializes global program variables and takes care of displaying and handling the command menus;

Local unit This unit contains general-purpose routines for interaction with the operator, such as error warning;

Remote unit Contains the routines which handle remote interaction, i.e. with the user using his terminal to dial into the local modem where the program is running;

User unit The final unit contains procedures which are executed during a remote connection as a result of /proc directives in the script file. These terms are explained in their appropriate chapters;

AuxInOut unit This unit is modified from the AuxInOut example in the Turbo Pascal manual [3] which takes care of the lower-level serial I/O through the communications ports.

Also, some files are used by the program:

Configuration file Allows certain parameters to be changed;

Script file Which controls the dialog with the remote user;

Password file Controls who accesses the system;

Log file Keeps a record of who did access the system.

These are all explained in great detail in later chapters.

Answer actually consists of a relatively unchangeable program core, supplemented by application-specific support code. By this I mean, that the function of the program can be changed completely without significantly altering the main program itself. The Answer program proper is a bulletin board system without a purpose.

But as the purpose of the program was already defined in terms of the problem posed in the “Introduction” chapter, support code has already been added to make the program useful. In the next chapter, “Upgrading Answer,” it will be shown how to add different functions.

USING ANSWER

The program is executed from the MS-DOS command line by typing `ANSWER`. The auxiliary files are normally expected to reside in the current directory.

† More precisely, the source code of each of the files can be found in .Pas files, and the object (compiled) code in .TPU (*Turbo Pascal Unit*) files. The resultant program, comprising all the compiled units sis the Answer.Exe file.

When the program is started, it initializes the Hayes-compatible modem which it expects to be in the port specified by the modemPort configuration parameter. The operator is then presented with a single menu, which has just three options.

Selecting “Answer calls” selects the normal course of operation for the program. In this mode, the program continually monitors the phone lines for another computer to dial into the system. When a connection is made, the message “call in progress” is blinked in green on the top line of the display. At any other time, the top line will show a message indicating that call monitoring can be ended by pressing the <Esc> key, which will return you to the menu. The program is intended to be completely self-sufficient in the monitoring mode, and should normally require no operator intervention.

The second option, “Initiate a call” is currently used for test purposes only—it can be used to ascertain whether the configuration parameters pertaining to modem operation have been set correctly.

Pressing <Esc> ends the program.

If an error occurs at any time during the operation of the program, a beep is sounded, an error message is displayed in red. Furthermore, if the system is currently entertaining a remote user, the connection is terminated. This is done for security reasons, so as not to allow users access to the system in an uncontrolled situation.

File-Transfer Extensions to Answer

As stated, the Answer program has already been given ‘extended’ code which answers the file-transfer problem. The extensions consist of /proc procedures (explained in the “Script File” chapter) in the User unit, which invoke the Kermit software to effect the actual file transfer, and a predefined script file named “ANScript.Txt” which presents the remote user with several menus to allow him to up- or download files. Passwords *are* used.

The user may download files if his privilege level is greater than or equal to ‘1’, from the directory specified by the constant downloadDir. Uploading of files is allowed if the user's privilege level is greater than or equal to ‘5’, into the directory specified by the constant uploadDir. There are also commands which allow the remote user to view these two directories.

Upgrading Answer

Many of the features of the Answer program can be changed quite easily by using a text editor and working on one of the auxiliary files. More advanced changes can be made by changing the actual program code. Answer has been written with this very upgrading in mind. This chapter attempts to explain all the details you need to know about the way in which Answer works that will allow you to customize and add your own features and commands to the program.

ADDING NEW COMMAND MENUS

As the program originally exists, there is only one command menu with three options in it. However, Answer has been written to allow many more, hierarchical, command menus. All the code and supporting routines for command menus resides in the Answer unit.

In Answer, each command menu has its own *menu ID.* The *main menu,* the one which is shown when the program is started, always has ID = 0. Implementing additional menus proceeds in two steps:

• First, you must write code that displays the menu's options. This is done in the ApplyMenu routine. Whenever the *menu* parameter in this routine matches the ID of your menu, simply use successive Writeln commands to display all the options.

• Immediately after this code, follow with a loop which scans the keyboard until a valid key is pressed, and then dispatch to your own routine. Refer to the code for the existing menu for an example.

Hierarchical menus can be implemented extremely easily—when the operator selects an item which should link to a lower-level menu, call GetMenuCmd with the menu ID of the next lower-level menu. The program will take care of returning you to the higher level in the menu structure. The number of levels which can be implemented is limited only by memory size.

Note that the <Esc> key is usually reserved to return to the previous menu without taking any action, as is done for the main menu.

ADDING CONFIGURATION PARAMETERS

This can be done by adding code to the Answer unit SetParameter routine. This routine will be called with the name of the configuration parameter in *param,* and its assigned value in *value.*

Configuration File

About the Configuration File

The configuration file allows you to set various parameters which affect the program's operation, without actually having to recompile the program. It is the file with the name 'ANConfig.Txt', in the current directory when the program was started. This name can only be changed by recompiling the program.

The configuration file consists of parameter assignments, with each one starting on a new line. These lines take the form of

```
parameter = value
```

where *parameter* is the name of a configuration parameter, and *value* is its assigned value. Both *parameter* and *value* are case-sensitive, i.e., upper- and lowercase characters are distinct. The program expects at least one space or tab character separating the equals sign ('=') on each side. If an error is detected while reading the file, an error message is displayed, and further reading is aborted. If more than one assignment is made to the same parameter, only the last one is significant. If no assignment at all is made to a particular parameter, the program supplies a default.

A comment, delimited by curly brackets ('{}'), may be inserted anywhere outside of the actual assignment, i.e., before the parameter or after its assigned value, but not inside the assignment. It is customary to start the file with a comment, identifying the author, the creation date and updates of the file.

Configuration Parameters

In the following list of configuration parameters, curly brackets ('{}') indicate that exactly one of the values in the list must be used. The value which is printed in boldface is the default.

`modemPort = {`**`com1`**`, com2}`

This specifies in which of the computer's serial communications ports the modem is installed.

`baudRate = {110, `**`300`**`, 1200, 2400}`

Specifies the default baud rate of the modem, or any rate at which the modem can operate. Note that during remote connections, it is entirely possible that the modem may switch speeds to adapt itelf to the calling modem.

`dialMethod = {`**`pulse`**`, tone}`

Specifies whether the modem should use the older pulse dialing method, which emulates a rotary dial-type phone, or the faster two-tone dialing system. Some older phone exchanges cannot use tone dialing. The default is the modem default, which is usually pulse dialing.

`modemSpeaker = {off, soft, `**`medium`**`, loud}`

The modem's speaker is usually turned on during dialing, so you may follow the status of the dialing operation. The speaker volume may be adjusted, or the speaker may be turned off, using this parameter. The default is the modem default, which is usually medium.

`timeOutLimit = `*`seconds`*

When a connection is made with a remote computer, and the system is waiting for a particular response from the remote user, timeOutLimit specifies the maximum number of seconds the system will wait. This way, if the remote user hangs up, or the connection is inadvertently lost, the system is not left waiting endlessly for a reply from a nonexistent user. The timeout feature can be disabled by setting timeOutLimit to zero. The default timeout limit is 60 seconds.

`logFile = `*`name`*

Name is the name of the file which is to be used to log when and which of your users log into the system. The default is no log file, in which case all the logging information is displayed on the screen instead.

Script File

About Scripts

The script file makes it possible to significantly change the remote interaction without changing a line of program code. However, if you wish, the script file does allow you to link your own customized Pascal routines to any part of the interaction.

The interaction proceeds in steps, called *screens.* For instance, when a remote user first logs on to your system, Answer automatically displays the screen named 'main'. Special commands, called *script directives* allow you to jump to different screens, to set up menus, or call external procedures. Screens can actually contain more information than a single screen on the computer—the beginning and ending of a 'script file screen' is determined by the *screen definition.*

A powerful feature of script files are *menu definitions.* Within a screen, it is possible to create fully automatic menus by using a series of simple commands. With menus, you can have the program interact with the remote user by jumping to different screens, depending on the user's response.

As in the configuration file, it is customary to start the file with a comment, identifying the author and the creation date etc. of the file. The comment can be made to appear the same by enclosing it in curly brackets ('{}'), but in fact anything outside of a screen definition is considered a comment. Using the brackets makes it clear to other people that the comment is not part of the remote dialog.

As the configuration file was initially set, the script file is the file with the name 'ANScript.Txt', in the current directory when the program was started. This name can be changed by setting the scriptFile parameter in the configuration file.

Script Directives

A script directive must always start on a new line in order to be recognized as such. All directives start with the backslash ('\') character to distinguish them from normal text. This also implies that whenever a backslash is the first character on a line, the program expects a directive to follow, and it will signal an error if it does not find one. Therefore, a line of plain text cannot begin with a backslash.

More often than not, the program expects a single space to separate any parameters following the directive. Note that the script directive names and the parameters are case-sensitive, so upper- and lowercase characters are distinct.

`\screen` *`screenName`* directive

This marks the beginning of the screen *screenName.* The program always starts the remote dialog with the screen named 'init'. Any lines of plain text that follow \screen, before the next \endscreen, are transmitted as is to the remote user.

`\endscreen` directive

This delimits the screen definition which was started by \screen. Any lines of plain text that follow \endscreen are ignored. When the screen is being displayed the program reaches \endscreen, the connection is terminated and the phone is hung up. This can be avoided by placing a \goto directive just before the \endscreen.

`\goto` *`screenName`* directive

Causes parsing of the current screen to be terminated, and continued at the screen named *screenName.*

`\wait` directive

Suspends parsing of the screen until the remote user presses the <Enter> key. Before using \wait, you should display a line stating that the user should press <Enter>.

If the user fails to respond within the period dictated by the timeOutLimit configuration parameter, the message 'timeout!' is displayed, and the connection is terminated.

`\proc` *`procedureName`* directive

Calls the external procedure *procedureName.* The routine actually called is DoProc in the User unit, with *procedureName* as a parameter. It is up to you to amend DoProc to recognize this name, and dispatch control to the appropriate routine.

`\getpassword` directive

Prompts the remote user to enter his user name and password, which are then checked against the password file. If they match, the current user privilege is set to the value entered in the password file. If the remote user fails to enter a correct user name and password in three attempts, the connection is terminated, meaning that anything following \getpassword is ignored.

If the user fails to respond within the period dictated by the timeOutLimit configuration parameter, the message "timeout!" is displayed, and the connection is terminated.

If logging is turned on, the user name, as well as the current date and time, are written to the log file. The name of the log file is preset to 'ANLog.Txt', but may be changed (or logging may be switched off) by setting the logFile parameter in the configuration file.

`\menu` directive

This is the most sophisticated of the directives. It starts a menu definition, which is terminated by the \endmenu directive. Within the definition, *menu directives* are used to define the appearance and effect of a menu, by describing the individual options the user has. These options, as they appear in the menu, are called *items.*

`\endmenu` directive

This terminates the menu definition started by \menu. Parsing of the screen is suspended until the user enters a character indicating his choice, followed by <Enter>. Only the first character of the response is significant. If the choice is not valid, the user is prompted again, until a correct response is received. If the chosen item was associated with a particular screen in the \item menu directive, anything following the \endmenu is ignored, and parsing resumes with the associated screen.

If the user fails to respond within the period dictated by the timeOutLimit configuration parameter, the message "timeout!" is displayed, and the connection is terminated.

`\item [key:` *`keyCharacter`*`] [screen:` *`screenName`*`] [privilege:` *`level`*`] [name:` *`itemName`*`]` menu directive

This directive causes the specified item to be added to the list of valid menu choices. Brackets ('[]') enclose optional parameters.

The key parameter defines the key that is used to select the item. KeyCharacter is the single character that selects the item, or the three-letter code "esc" to indicate the <Escape> key. <Escape> key should be used to return the user either to the next-higher level menu, or to the main menu. Although a key need not be associated with the item, this simply makes the item unselectable, and serves no known practical purpose.†

The screen parameter causes parsing of the script file to continue at the screen entitled *screenName* if this item is chosen. If the screen parameter is not included in the item definition, parsing will resume following the next \endmenu directive.

Privilege allows you to lock out users who do not have sufficient privilege from choosing the item. Specifically, if their privilege is lower than *level,* the item will not be displayed and will be unselectable.

Finally, the name parameter specifies the name of the item, as it will appear in the menu. If no name is specified, the item will not appear in the menu, but it will still be selectable.

Warning: *Everything* following "name:" is considered part of the name, so this must be the last parameter in the \item directive.

† Actually, the item should still be selectable with <Cntl>-A, but it is not advisable to use control- or Alt-key combinations for menu items

`\default [key: `*`keyCharacter`*`] [screen: `*`screenName`*`] [privilege: `*`level`*`] [name = `*`itemName`*`]` menu directive

This is identical to the \item directive, except that the declared item will also be chosen if the remote user types just an <Enter>, without making a selection. This item should be the safest course of action, e.g. a help screen. Note that there is no need to define the item twice, i.e. once as an \item, and once as \default. If the user does not have sufficient privilege for the default item, the program will act as if there was no default item, i.e., parsing will resume following the \endscreen directive.

`\timeout [key: `*`keyCharacter`*`] [screen: `*`screenName`*`] [privilege: `*`level`*`] [name = `*`itemName`*`]` menu directive

This is identical to the \item directive, except that the declared item will also be chosen if the remote user does not respond within within the period dictated by the timeOutLimit configuration parameter. Note that there is no need to define the item twice, i.e. once as an \item, and once as \timeout. If the user does not have sufficient privilege for the timeout item, the program will act as if there was no timeout item, i.e., parsing will resume following the \endscreen directive.

EXAMPLE

The following is an example of part of a script file, as it appears in the presupplied sample file:

```
\screen mainmenu
Please select a command from the menu.
\menu
\item key: d screen: transmit privilege: 5 name: Download
\item key: u screen: receive privilege: 3 name: Upload
\default key: h screen: help name: Help
\timeout key: esc screen: quit name: Quit
\endmenu
\endscreen
```

A user with a privilege level of 4 will see the following when this screen is parsed:

```
Please select a command from the menu.
(u)Upload  (h)Help  (q)Quit
```

The user has too low a privilege for the ‘download’ option. If the user enters the character shown with one of the options, that action is executed. For instance, if the user types an ‘h’, the Help screen is executed. Control is also passed to that screen when the user presses Return, since Help is the default item. If nothing happens within the prescribed timeout period, control is passed to the Quit screen.

Password and Log Files

Password File

The *password file* is the file containing the user names, passwords and privilege levels for all the users who are allowed access to the system. If logging is on, the user name can be used to determine which of your users are accessing the system. It is not necessary to use passwords on the system, but they are invoked with the \getpassword script directive (cf. the "Script File" chapter). If you don't want to use passwords, you don't need to create a password file. The password file is the file with the name 'ANPass.Txt', in the current directory when the program was started.

A *user name* is a name (or call sign) you assign to a particular user or installation that you want to be able to gain access to the system. User names are generally not secret. The *password,* however, which should be unique to each user, is. The *privilege level* controls what level of access you allow a particular user, with higher numbers implying more access. For instance, you might not allow users with a privilege level lower than '5' to upload files to your system, though they might be allowed to download files.

The password file consists simply of successive password entries on each new line, which are of the syntax:

```
    userName password privilege
```

Note that even if you don't use privileges, you must assign one, or the program will signal an error.

Warning: The password protection given by this system is only rudimentary. For instance, anyone with access to your installation may easily read, add or delete passwords from the file.

Warning: If you allow users to up- or download files to or from your system, make sure they do so only in an especially assigned subdirectory. Otherwise, users with malicious intent may be able to locate and amend the password file.

Log File

The *log file,* if one has been specified by the logFile configuration parameter, is updated with a log entry whenever a user attempts to log onto the system. These entries are single lines of the form

```
    date time {userName, timeout on login, access denied}
```

where *date* (in day/month/year format) and *time* indicate the moment when the user attempted to log in. If the user did in fact log in by correctly entering his password, the user name ends the entry. Otherwise, the message "timeout on login" is shown if the user failed to gain access within the timeout period, or "access denied" if the user did not enter a correct user name and password in three attempts.

Note that if passwords are not used (i.e. there is no \getpassword directive in your script file), there is no logging either.

Bibliography

[1] Electronic Industries Association, *EIA Standard RS-232-C Interface Between Data Terminal Equipment and Data Communications Equipment Employing Serial Binary Data Interchange,* October 1969

[2] *MS-DOS Kermit User Guide,* C. Giacone, F. da Cruz, J.R. Doupnik, Trustees of Columbia University, 1988

[3] *Turbo Pascal Reference Guide, Version 5.0,* Borland International, 1988

[4] International Standards Organisation, *Open Systems Interconnection-Reference Model*

Index