Remote file access over low-speed lines

James Mortimer Hague

A thesis submitted for the degree of Doctor of Philsosphy

University of Kent at Canterbury

July 1988

Abstract

A link between microcomputer and mainframe can be useful in several ways, even when, as is usually the case, the link is only a normal terminal line. One interesting example is the 'Integrated application', which divides a task between microcomputer and mainframe and can offer several benefits; in particular, reducing load on the mainframe and permitting a more advanced user interface than possible on a conventional terminal.

Because integrated applications consist of two co-operating programs, they are much more difficult to construct than a single program. It would be much easier to implement integrated applications concerned with the display and/or modification of data in mainframe files if the microcomputer could confine its dealings with the mainframe to a suitable file server. However, file servers do not appear practical for use over slow (compared to disc access speed) terminal lines.

It was proposed to alleviate the problems caused by the slow link with *extended file operations*, which would allow time-consuming file operations such as searching or copying between files to be done in the file server. It was discovered after attempting such a system that extended file operations are not, by themselves, sufficient; but, allied to a record-based file model and asynchronous operations (i.e. file operations that do not suspend the user program until they complete), useful results could be obtained.

This thesis describes FIAP, a file server for use over terminal lines which incorporates these ideas, and MMMS, an inter-application transport protocol used by FIAP for communication between the microcomputer file interface and the mainframe server.

Two simple FIAP applications are presented, a customer records maintenance program and a screen editor. Details are given of their construction and response time in use at various line speeds.

To Heather

Isn't this just what
you've always wanted ?

Acknowledgements

First and foremost I must thank my supervisor, Bob Eager, who has been an inexhaustible source of suggestions, ideas and a wide range of information. His work making Kermit available to computer users around the campus lies at the root of this thesis.

Many others have helped with explanations, facilities or just having the tolerance to listen to my ravings. Tony Varlese, Dave Bisset and Afzal Bhatti come to mind immediately, but there were many others. I should also like to thank Keith Petersen for his work maintaining the SIMTEL20 microcomputer archives.

My parents have been an invaluable source of support, both financial and otherwise, and will doubtless be relieved at the prospect of their eldest son finally finishing a student career that turned out to be rather longer than they (or I) had anticipated.

Finally, a very big thank you to Heather, for all her support, encouragement, prodding and patience. I can't imagine how she managed to put up me. Thanks, Possum.

This work was supported financially by a studentship from the Science and Engineering Research Council (SERC).

Table of Contents

List of Tables

List of Figures

Chapter One

Introduction

This chapter introduces microcomputer to mainframe communication, beginning with a very brief history of the microcomputer. The motivation behind making such a connection is examined, and the various mechanisms currently employed are highlighted. The chapter concludes with an overview of the rest of this thesis.

1.1. The Microcomputer — a short history

The early 1970s saw Large-Scale Integration technology advance to the stage where it became possible to produce microprocessors at low cost [Lapidus72]. Initially aimed mostly at the then-burgeoning calculator market, several processors made their way into amateur computer designs. By the middle of 1975, several systems were available to the hobbyist in kit form†.

The first microcomputers aimed outside the hobbyist market emerged in late 1977, with the launch of (amongst others) the Apple][and the Commodore PET. These were initially directed at the home market, but were soon being used in a variety of applications (e.g. small company accounts, experimental data logging) where computerisation had not previously been economically viable. The advent of the first spreadsheet program, VisiCalc, which one reviewer [Ramsdell80] noted was 'responsible for the sale of entire systems', opened up new markets among finance professionals who now found a microcomputer an indispensable tool.

The advent of the IBM Personal Computer at the close of 1981 was felt at the time [Williams82] to confirm the arrival of the microcomputer as a business tool and also introduced a new generation of microcomputers based around the new 16-bit microprocessors then emerging. The IBM PC has gone on to become an industry standard, with many manufacturers producing compatible machines. Other manufacturers have competed successfully with machines that are not 'IBM compatible' but technically superior and/or simply cheaper.

† A look back at the genesis of the amateur computer can be found in [Libes78]. [Ogdin74] gives an overview of the more common microprocessors available by early 1974.

The standard IBM PC, in the guise of cheap compatibles, is now available alongside audio and television equipment as an item of consumer electronics. Estimates of how many microcomputers of equivalent class or above are currently extant in the world vary between 8 and 20 million, and at the time of writing (early 1988) the combined sales reported by the top four volume producers of microcomputers are in the region of 1 million per month.

Many of these microcomputers are in use within organisations that also possess mainframe computers. Indeed, it is not uncommon for a microcomputer to sit side by side with a terminal attached to a mainframe system. This naturally leads to the idea of linking the two in some fashion; motivations and means for achieving this are examined in the next section.

1.2. Reasons for connection and connection mechanisms

1.2.1. Why ...

The provision of some form of communication link between microcomputers (micros) and mainframes has become increasingly common. The gains from such a link can be classified into two main areas — information access and operational benefits — and are outlined below.

1.2.1.1. Information access

A micro-mainframe link can improve the quality and/or quantity of information available to the end user. These improvements may be divided into at least the following three categories.

1. *Access to centrally-held information.* In a business environment, extracting information from the corporate database for further analysis on the micro, or for inclusion in reports (or both) — for example downloading monthly sales figures into a spreadsheet for analysis and display in a variety of forms. A more scholarly example is extracting entries from a master bibliography for inclusion in a paper under preparation on a micro. Libraries of useful programs for micros may be held centrally on a mainframe. Of course, the traffic need not necessarily be all in one direction; updating the corporate database with departmental information is another possibility.

2. *Access to mainframe networking and communication facilities.* It is not unusual for a large computer to have facilities for communicating with other installations which are unlikely to be available to the micro. Electronic mail and conferencing, and the ability to search remote databases and/or public information sources are two possibilities.

3 *Access to mainframe tools.* The power of, say, mainframe statistical analysis tools may greatly exceed anything available on the micro for analysing data generated on the micro. Or one aspect of some engineering calculations may need substantial processing power to be completed in a reasonable time.

1.2.1.2. Operational benefits

Even if it does not enhance the quantity or quality of information available, a micro-mainframe link can also be of value to the operation of both the micro and the mainframe.

1. *Savings in equipment.* If access to a mainframe is to be provided, then unless either specialist terminal facilities are required or it is expected the terminal will be in constant use, use of terminal emulation software on a microcomputer which is free for local use at other times (or indeed already present) may offer worthwhile economies in equipment cost.

2. *Use of specialised hardware.* Mainframe peripherals frequently include items such as high-speed or high-quality printers, plotters and other devices not available to the micro user.

3. *Use of mainframe backing store.* This offers two advantages — size and safety. Mainframe backing storage capacities are usually orders of magnitude larger than microcomputer storage, and probably more importantly are much more likely to be subject to proper backup and archival procedures — microcomputer users are notoriously bad at keeping full backups of their data.

4. *Use of local computing power to decrease load on the mainframe.* An 'integrated application', splitting the job between mainframe and micro, may allow the micro to take over aspects of the task (for example, updating the screen and moving the cursor) that make heavy demands on mainframe I/O bandwidth. Note that such an application may well also provide one of the benefits listed in §1.2.1.1.

5. *Improvement in user interface*. A suitably-programmed microcomputer may offer user interfaces not possible with a conventional mainframe terminal. In particular, a microcomputer application may make use of windows, pop-up menus, mice and so on.

1.2.2. ... and how

There are a variety of possible link options. In rough order of complexity, these may be outlined as follows.

1.2.2.1. Terminal emulation

Software and in some cases hardware is added to a micro to allow it to behave as a terminal (as far as the mainframe is concerned). The package usually emulates some popular type of terminal such as the DEC VT100 or IBM 3270.

The emulation basically involves recognising incoming commands from the mainframe affecting the screen (including sequences which, for example, clear the screen or position the cursor). The emulator must also monitor the keyboard and take the appropriate action on keystrokes. For most keys this will just mean sending the indicated character to the mainframe; some keys, such as function keys, may required a command sequence to be sent, and possibly local action as well.

Using terminal emulation, the mainframe controls the interaction completely and is not usually aware it is communicating with anything other than a standard terminal.

1.2.2.2. File transfer

This allows files held on either the micro or the mainframe to be send over the link and saved in the filespace of the receiving system. This may be accomplished by mechanisms ranging from simply allowing the terminal emulation program to save incoming characters into a file and transmit files as if they were being typed on the keyboard, up to using special programs to send the file(s) using error-correcting file transfer protocols.

During file transfer, the micro and the mainframe co-operate on a roughly equal footing.

1.2.2.3. Using the mainframe as a server

Here the micro treats the mainframe as an extra device, typically a file system device, and issues requests for files to be opened and closed, data to be read or written and so on. These files held on the mainframe file store may be either held as normal mainframe files (*file server*), or stored in space reserved for the micro using the file storage methods employed by the micro filing system (*virtual disc server*).

This is in one sense the inverse of terminal emulation, in that the micro controls the interaction completely, and is not necessarily aware it is using mainframe facilities rather than a local facility.

1.2.2.4. Integrated application

An integrated application splits the job in question between the mainframe and micro, in proportions appropriate to the job in hand. Both sides are aware of the presence of the other. For example, consider a screen editor split such that the micro looks after all display functions but leaves all actual editing to the mainframe. For a given application, this means that only the minimum necessary quantity of data need be shipped over the line.

The extent to which either side of the application can be said to control the other depends upon the application in question. Unlike the above three categories, integrated applications cannot be generally characterised as either one side or the other (or neither side) in control.

1.3. Current practice

1.3.1. Connection

By far the most common means of connecting a micro to a mainframe is to use a standard serial terminal connection. This terminal connection may be provided by any of the usual means — directly connected, via a dial-up line using a modem, connected via a wide-area network or whatever.

Modern microcomputers, almost without exception, include an asynchronous serial port as standard. This, together with appropriate software, is sufficient to communicate with almost any type of contemporary mainframe. The important exception to this is IBM mainframes†, where terminal lines use

† and, of course, so-called IBM plug-compatible mainframes produced by several competing corporations.

synchronous serial communication. Additional hardware is necessary to connect a micro successfully in this case; [Madron87] summarises the possible approaches.

Synchronous or asynchronous, serial communication via standard terminal lines does not give a high data transfer rate — the fastest lines offer rates in the order of 1000 characters per second full-duplex. This can be rather a bottleneck if the volume of data transfer is much above that normally required by terminal operations. Several manufacturers now offer products [Haugdahl84] allowing micros to be connected to high-speed local area networks and thus to communicate with the mainframe using data transfer rates comparable to those achieved by micro floppy/hard disc subsystems. Such systems will undoubtedly become more prevalent in the future; at the time of writing the low-cost (not infrequently free) nature of terminal line connection ensures its dominance. As users get used to being able to connect to mainframes and become aware of the problems of the low data transfer rates then the demand for high-speed connections will increase. In the meantime, terminal line connections predominate. As the work described below concentrates on making effective use of these links, the following discussion concentrates on the use of terminal rather than higher-speed links.

1.3.2. Terminal emulation

Once connected, some form of terminal emulation software is required to handle screen updating and keyboard monitoring as outlined in §1.2.2.1. The type of terminal emulated is commonly one of the DEC VT100 terminal family for asynchronous lines or one of the IBM 3270 family for synchronous lines. In either case a wide choice of products is available both from the mainframe manufacturers and others.

Despite being the easiest form of micro-mainframe connection to implement, terminal emulation is not without its problems.

— Character set translation. The author is not aware of any modern microcomputer that does not use the ASCII character set†. When connecting to IBM mainframes, which use EBCDIC, translation between the two must be provided. The ASCII and EBCDIC character sets overlap, but do not coincide completely, leaving the problem of how to translate characters that do not have an equivalent

† This has not always been the case. Some early microcomputers aimed at the home market did use custom character sets and encodings (e.g. Sinclair ZX-80).

representation in the other character set.

— Incomplete emulation. Terminal emulators very rarely provide a full emulation of the target terminal, for one of two reasons

(i) Hardware limitations. To be a true emulation, the micro should have an identical keyboard to the target terminal, and also possess screen facilities that are at least equivalent to the target terminal. In practice, neither of these are very common and so the emulation must just do the best it can. For example, a VT100 emulator running on a micro unable to display characters in **bold** might choose to display them in a different colour (if colour is available) or failing that in reverse video. Equally, emulating a terminal with 10 function keys using a micro keyboard with only 4 function keys will involve some compromise.

(ii) Software limitations. Writing a terminal emulator that comes as close as possible to the target terminal as is possible using a given micro may simply just not be worthwhile. The programming effort involved in adding just those last few functions may easily be as much as was expended in writing the rest of the emulator. If the lack of those functions is going to inconvenience only a small minority of users then the manufacturer will think twice before making the effort. Another possibility is that the changes necessary to allow implementation of a function would make the rest of the emulator less pleasant to use, perhaps slower or less robust. To illustrate, consider the example given above where the micro cannot display characters in **bold** using its normal character display mode. However it might be possible to come close by using a graphical display mode and selecting a thicker font than normal to display bold characters. This could well significantly slow down the rate at which all characters could be displayed; if only a minority of users of the emulator need to be able to distinguish bold and reverse video characters while the majority insist that characters be displayed as fast as possible the emulation may well stay incomplete.

— Keeping up with incoming data. A terminal emulator may find it impossible to cope with incoming data as fast as a real terminal, for a variety of reasons. If some form of flow control is not available, the emulator may drop characters in certain circumstances.

Some terminal emulators offer facilities beyond emulating a given terminal. The advent of micro-computers and workstations with high-resolution graphical screen capabilities has popularised user interfaces based on windows, pointers and mice. [Pike84], [Collinson86] and [Parker87] all describe systems where multiple terminal sessions may be conducted simultaneously in different windows, communicating with multiple terminal processes on the host mainframe using protocols that multiplex input and output from each process onto the one terminal line.

1.3.3. File transfer

Most terminal emulation programs provide facilities for recording data received onto a local disc file, or sending a file as if it was being typed at the keyboard. These can be used as a rudimentary means of file transfer. To transfer a file from a mainframe to the micro, the user requests the terminal emulator to record all received data into a file and then instructs the mainframe to list the file to the terminal. To send a file, the user requests the mainframe to enter all incoming data into a file (by, for example, starting an editor and setting it to an 'enter text' mode), and then uses the terminal emulator to send the file.

1.3.3.1. Possible problems

In practice, the above method of file transfer is not often used. A file transfer protocol is used instead; such protocols address one or more of the following problems encountered transferring files from micro to mainframe.

1. Terminal line limitations

   (a) Reliability. Transferring a file from one system to another is of little or no use if the file is corrupted in the process. This corruption may be due to one or more of a number of factors — noise on the line, input buffers overrunning and dropping characters and so on.

   (b) Transparency. It simply may not be possible to persuade a terminal line to pass a particular byte unchanged. Mainframe terminal lines frequently insist on accepting 7-bit ASCII characters only, using the eighth bit as a parity bit. Other characters cause problems too — on many systems control characters have special meanings (e.g. Control-C aborts the current program, Control-S inhibits the sending of data and so on).

(c) Input format. While some operating systems will happily accept input one character at a time, others prefer to deal in records. Input is buffered, and only passed on when some end-of-record character occurs (due to its synchronous nature, communication with IBM mainframes works this way). If the last character in the file is not one of the set of end-of-record characters ....

2. Data representation problems

(a) Text encoding. The possibility of problems with ASCII to EBCDIC translation (and vice-versa) has already been mentioned in §1.3.2. In addition, different systems have different ways of representing a line of text, ranging from a count followed by the characters on the line to the characters terminated by a Carriage-Return/Line-Feed sequence. Some systems attach meanings to other control characters.

(b) File formats. Some systems support a wide range of possible file types, frequently record-oriented, while others support only bytestreams.

(c) Data format. The pattern and number of bits used to represent characters, integers, floating-point numbers etc. vary wildly from machine to machine.

[da Cruz87] contains a more detailed discussion of some of the above issues.

1.3.3.2. Some current micro-mainframe file transfer protocols

Any number of file transfer protocols have been designed for microcomputer use, all assuming use over asynchronous serial lines. The details of some have been published and the protocol placed in the public domain to encourage widespread implementation. Others are proprietary, their designers preferring to keep their details secret. The success of such protocols is dependant on the fortunes of their suppliers, and they have not by and large gained widespread acceptance.

The main features of four (public) protocols are compared below. The protocols compared are Xmodem, Kermit, Compuserve B and Zmodem. Details of their workings are **not** given; these are compared in §2.1.

Before proceeding, it should be noted that comparisons are based on the original protocol definition using default options; in the case of the first three, most of the deficiencies highlighted are at least alleviated

by use of options or extensions. The mechanisms used by these options and extensions are not covered here, but again postponed to §2.1.

1.3.3.2.1. Xmodem

Xmodem [Christensen82] is the oldest of the four protocols discussed in this section. It was designed in 1977 and extensively revised in 1979 by Ward Christensen, a BBS Sysop† in Chicago, to enable the CP/M community to exchange files via modem. It is also known variously as *Christensen protocol*, *MODEM2*, *CP/M Users Group* and *TERM II FTP 3*, and is claimed by its supporters to be the most widely implemented microcomputer file transfer protocol.

Because Xmodem was originally designed for use between two microcomputers, it assumes a completely transparent link. Also, as CP/M files are made up of one or more 128 byte blocks, Xmodem transmits data in blocks of 128 bytes and can thus introduce unwanted extra characters at the end of a file. Xmodem does not address any data representation problems.

Xmodem's main strength is its simplicity. Criticisms are that it is susceptible to line errors and, because it assumes total link transparency, impossible to connect to many mainframe systems. Despite this, it is in widespread use by users of those mainframes that can meet the link requirements.

As Xmodem's failings have become apparent, it has spawned a number of derivatives, each of which corrects one or other perceived defect. Xmodem-CRC [Byrns85], Ymodem [Forsberg87], WXmodem [Boswell86] and MEGAlink [Meiners87] are some of these derivatives. In particular, Xmodem-CRC offers vastly improved error detection for transmitted data, while WXmodem provides greatly improved throughput.

1.3.3.2.2. Kermit

Kermit [da Cruz87] was designed in 1981 specifically for micro-mainframe file transfers. Consequently it does not require a transparent link, only that all printable ASCII characters and one (selectable) control character be passed without corruption. Uniquely amongst the protocols reviewed in this section it

† A Bulletin Board System (BBS) is a dial-up system which provides a public forum for messages to be exchanged. A typical BBS will also allow files to be transferred to or from the system. Some modern systems allow different BBSs to exchange messages, providing an electronic mail service. The *Sysop* is the system operator.

can be used with IBM mainframes without special provision. Public-domain implementations are available for nearly all current mainframe and microcomputer systems.

Kermit also partially addresses two text representation issues. The protocol specifies that all communication must take place using ASCII characters, and further requires that the end of a line in a text file be indicated by a Carriage-Return/Line-Feed sequence. Systems such as Unix which use a different convention (in the case of Unix, a single newline character) substitute as appropriate. Files can be specified as binary to avoid this translation.

Uniquely among the protocols discussed in this section, the Kermit protocol specifies a *server* mode. Server operation (and the ability to issue server commands) is an optional feature of the protocol, but a widely-implemented option. When a Kermit program is placed in server mode, it stops accepting commands from the keyboard and receives further instructions via packets from the Kermit program at the other end of the line requesting various functions. If many files have to be transferred, using server mode to allow the local Kermit to control the remote Kermit program saves having to issue the relevant commands to both.

The functions available in server mode, as well as those requesting file transmission or reception and the command to quit server mode, include many file management commands. These can be used to copy files, rename files, list directories, shut down the server and log out, and so on.

Kermit is less susceptible to line errors than Xmodem, due to better error detection facilities, but is slower because of the overhead of transmitting all control characters encoded as printable characters. Like Xmodem it has been subject to many improvements since its original design, but as the original design allowed for future extension via a capability negotiation mechanism, extensions officially approved by its designers at Columbia University have been neatly integrated within the Kermit specification. As with Xmodem, these improvements include improved error detection† and throughput.

† Although the improved error detection facility was present in the first published descriptions of Kermit [da Cruz84], the designers have confirmed that it was an extension to the original design [da Cruz88].

1.3.3.2.3. Compuserve B

Compuserve is a large public-access system in the USA that offers subscribers electronic mail facilities and public forums for a variety of (usually computer-related) discussions. One of its attractions is its large library of public-domain software available for downloading. It was with this in mind that the Compuserve B [Compuserve85] protocol was designed‡. As a consequence of being designed with one host system in mind, the protocol requires a terminal line as transparent as those used to access Compuserve. The terminal line must be able to pass eight bit characters, and all but 7 control characters. As with all control characters in Kermit, these 7 control characters are always encoded — no effort is made to establish whether the line really needs them to be encoded or not.

Compuserve B also defines a preamble sequence which, if incorporated into a terminal emulator, will cause file transfers using the protocol to be automatically invoked. This greatly enhances its ease of use. Instead of having to invoke the mainframe file transfer program and then separately start the microcomputer file transfer program, the mainframe program can start the microcomputer program by itself, the user only needing to issue the one command to start file transfer.

Compuserve B is less susceptible to line errors than Xmodem, but not as robust as Kermit. To expand somewhat on this: all three detect errors in transmitted data using a single character check. However, spurious acknowledgements of packet receipt (ACKs) or requests for retransmission (NAKs) can be generated easily by line noise when using Xmodem, which employs single character ACKs and NAKs. Compuserve B uses single-character NAKs, but two character ACKs, so line noise is less likely to generate spurious ACKs. Kermit sends ACKs and NAKs as full-blown packets complete with check, a five character sequence in all — spurious ACKs and NAKs are therefore very unlikely. It is notable that the improved error detection extensions in Xmodem and Compuserve B do not apply to ACKs and NAKs, so leaving the extensions just as vulnerable as the original protocols in this respect.

The protocol does distinguish between binary files and text files. Text files are guaranteed to be in ASCII, translation being performed as necessary, but no standard end of line convention is defined.

‡ To satisfy the curious, there was a Compuserve A protocol. It apparently had design flaws that necessitated its replacement — no description of it has been found in the literature.

As implied above, an extension to the Compuserve B protocol has recently been promulgated [Compuserve87], offering the usual improvements in error detection and throughput.

1.3.3.2.4. Zmodem

Zmodem [Forsberg87b] is the newest of the protocols reviewed here, and by far the most technically sophisticated. The name implies it is yet another mutation of Xmodem, though this is not in fact the case. It was developed to allow efficient operation when used over packet-switched networks, satellite channels, error-correcting modems and other links where the protocols above perform poorly, as well as to make more efficient use of communication bandwidth than possible with the above protocols. Attention was also paid to making implementation possible on a wide variety of microcomputer and mainframe systems. Unlike the above protocols, Zmodem requires a full-duplex link, although it is careful to place minimum demands on the reverse channel for reasons of efficiency when using some modern high-speed modems.

Zmodem currently requires the link passes 8-bit characters, although the design does allow for future extension to cope with 7-bit only links. The link must also be able to pass the ASCII CAN character, but all other control characters may be encoded to printable characters by the transmitter. The specification only requires 3 control characters always be escaped.

To aid dealings with text files, Zmodem defines two valid end of line sequences and permits transmitters to order their replacement with the local end of line convention (there is the unspoken assumption that text files are transmitted using the ASCII character set). The specification also defines a preamble sequence à lá Compuserve B.

Rather than use a simple single character check, Zmodem protects all transactions with a multi-character check of superior capability, as used in options and extensions to the above protocols (by default, the check is of the same effectiveness but optionally an even more effective one using more characters may be used), making it by far the most reliable of the protocols surveyed here. It also by default employs the methods used by the above options and extensions to improve throughput, and claims better performance than for the other three even using their throughput improvements, at least according to the performance figures published in the specification. However the increase in implementation effort (due to the increased

complexity) means that it has yet to become as widespread as the other three protocols.

1.3.4. Servers

The use of remote file servers connected over high-speed local area networks (LANs) has long been practised in both the mainframe [Dion82] and microcomputer worlds [Anada84, Dellar82]. Microcomputer LANs [Haugdahl84, Jesty85] have tended to use proprietary networking technology in an attempt to keep costs down, though as the cost of connection to, for example, Ethernet [Mier84] continues to decrease so interfaces to such networks have multiplied. A recent trend has been the use of mainframe-based file servers for microcomputers, either by providing support for mainframe remote filing systems (e.g. Sun Microsystems' IBM PC interface to its Network Filing System [Sun86] PC-NFS), or by providing mainframe support for microcomputer network architectures (e.g. Alisa Systems' AlisaTalk Appletalk interface for DEC VAX [Alisa87]).

With regard to the use of terminal-speed links to implement file servers, [Pechara81] reports implementation of a block device driver for a microcomputer implementation of Unix which read and wrote blocks over a 300 baud line. Transferring 6 blocks (each block containing 512 bytes) took in the order of two minutes. It is perhaps not surprising that the author has been unable to find reports of similar work.

The above discussion on servers has concentrated on a mainframe providing a service to the micro, the micro remaining in control. The use of the processing power of the microcomputer to provide a screen service to a mainframe application, attempting to minimise cross-link traffic can also be considered as an example of a server. A trivial example is the so-called 'intelligent' terminal, providing powerful screen management functions (and in some cases data entry facilities) coupled with software on the mainframe capable of making use of these functions to carry out screen updates in the most efficient manner (e.g. [Arnold84]). A more sophisticated example is Joint Network Team Simple Screen Management Protocol (JNT SSMP) [Hunter82] which allows the terminal to make cursor movements independently of the controlling mainframe via the mechanism of the *cursor box*. Briefly, this allows the terminal a certain amount of autonomy when cursor movements are requested. The mainframe application defines an area, the cursor box, within which the cursor may be moved without the mainframe having to be involved; it is informed of the new cursor position when the user attempts to move the cursor outside the box or issues some command

that must be acted upon by the mainframe. Text may be typed into the box under local control and, if so, it is forwarded to the mainframe at the same time as the new cursor position. This both improves the speed of cursor movement and screen updating during text entry, and reduces the load on the mainframe.

1.3.5. Integrated applications

Integrated applications to date have been chiefly concerned with using local processing power to present an attractive user interface; [McManis85] describes the advantages of such an approach to remote conferencing systems.

Several screen editors have been built that adopt this approach. [Frazer79] reports a screen-based front end *s* for the portable line editor *edit* [Kernighan76] which, although initially conceived as running on the same machine as the *edit* process, nevertheless suffers little degradation if connected to a remote *edit* process over a terminal line.

The Blit terminal [Pike84] is a powerful workstation in its own right, with a high-resolution windowing display backed up with a multitasking operating system and a protocol that multiplexes communication between processes in the terminal and processes in the Unix mainframe over the single connecting terminal line. The Blit system has several facilities provided by integrated applications, notably the debugger *joff* [Cargill85] and the screen editor *jim* [Pike84] and its successor *sam* [Pike87]. The Unix mainframe handles the 'meat' of each application, while the process in the terminal provides a modern graphical interface to the user. In this respect they are comparable to the window-based front ends provided for several common Unix tools (e.g. *shelltool*, *dbxtool* [Sun86b]) when running on the Sun workstation, although, of course, the latter do not communicate over a terminal link.

[Madron87] gives an overview of the *Remote Access Facility* (RAF) supplied by Datability Software Systems. This allows a standard IBM PC class micro access to files held on a remote DEC VAX mainframe running VMS, when the two are connected by either a terminal link or an Ethernet. It also provides facilities to allow the PC to execute VAX-based programs or call VAX subroutines. The VAX file system appears to the PC user as one or more MS-DOS disks, and the system allows end-users to access VAX applications as if they were local PC programs. It should be noted that RAF does not appear to offer the

possibility of writing true integrated applications where both sides act independently. A similar system, Möbius [FEL88], goes further, providing full task to task communication and allowing users the choice of working from the VAX or PC command line. Because such systems integrate the VAX file system with the local file system, they can easily fall foul of a slow link if, for example, the MS-DOS *copy* command is used to copy VAX files (the file would be brought down the link to the PC and then sent back again). In practice, the manufacturers of both RAF and Möbius discourage use over links other than high-speed local area networks.

1.4. Reflections

Using an ordinary terminal line to connect microcomputers to mainframes is likely to remain important for some time to come. In particular, data transfer rates achievable over the public telephone networks (PSTN) have a finite upper bound [Martin76], and even the fastest modems available currently only offer rates comparable with terminal lines [Humphrey88]. The overall aim of this work has been to make the best use of these links, to realise the benefits outlined in §1.2.1, in particular those of improving the user interface and reducing mainframe load.

As noted in §1.2.2.4, integrated applications offer the best method of making optimal use of the data transfer rates available. However Pike observes [Pike84] that writing one program in two communicating sections is much harder even than writing two separate programs, as one must deal not only with the two programs but also with their mutual and sometimes complex interaction. In some cases this complexity may be unavoidable, for example if the service required of the mainframe is some application-specific form of calculation. On the other hand, if all that is required is access to the mainframe filestore, then if some form of file server and interface were available, the programming effort would revert to that involved in writing just the one program.

Again, Pike makes the piquant observation that a typical terminal link is not fast enough for file operations, citing as an example the case of carrying out a search from within an editor. The editor would have to read the file across the link until the search was satisfied. At a line speed of 120 characters per second, a worst-case search on a 30k file would take in the order of 4 minutes. This is clearly not accept-

able.

The nub of the problem here is the need to ship data over the link in quantity. The intention of this work is to see if this problem can be mitigated sufficiently to make a file server practicable by providing operations such as searching in the server. Whilst this can in no way be a universal solution (obviously for any given set of server capabilities, applications exist that will still require substantial data transfer), it is hoped that a suitable choice of otherwise problematic operations (henceforth referred to as extended operations), such as searching, would enable a server to be useful to a significant range of applications.

1.5. Initial work and overview

Initially two extended operations were identified as useful, namely searching a file and copying between files. A start was then made on implementing a remote file access protocol incorporating these two operations.

As mentioned above (§1.3.3.2.2) the Kermit file transfer protocol includes a provision for server operation, in which not only file transfers can be requested, but also many file management operations. To investigate the ideas outlined in the previous section (§1.4), the basic Kermit server protocol was extended to allow file access operations, including the two extended operations mentioned. Work was started implementing the extended protocol between a Unix system (server) and an IBM PC or equivalent system on which user programs would run requesting file access functions.

An interface to the PC serial port capable of operating at the maximum available line speeds was constructed, together with a terminal emulator using that interface (Appendix A). An existing Unix Kermit program was modified to include the file access functions in its server mode and an interface program written for the PC that made these functions available to user programs on the PC, translating function calls into protocol transactions and returning results as appropriate.

Once this system was sufficiently complete for experience to be gained in using it (chiefly in writing test programs), several flaws and limitations became apparent.

— Communication speed. Although robust, Kermit does not make very good use of communication bandwidth, only being around 34% efficient [Tootill87].

— Program interface. With the exception of the extended operations, all file operations requested by user programs were implemented as synchronous operations, with control not being returned to the user program until the function had completed. If large amounts of data had to be moved across the link, or (worse) the server crashed, this could result in delays in the order of a minute or more before control was returned to the user program. This was not felt to be acceptable.

— Layering confusion. The protocol crossed several boundaries between the layers defined in the ISO OSI seven-layer model [Zimmerman80]. This had two major repercussions:

  (i) Alterations or extensions to the functionality provided meant modifications right down to packet level, requiring a considerable amount of work.

  (ii) The system was tied to the Kermit-derived protocol. As noted above, efficient use of the communication bandwidth was desired. Modification or replacement of the protocol affected everything — implementation would have to start almost from scratch.

— The bytestream file model used proved inadequate. All but the simplest file operations required several interactions with the server, piling delay upon delay. In particular, imposing any form of record structure over the bytestream except sequential fixed-length records could involve an unacceptable number of file operations to accomplish anything.

Accordingly, the system was redesigned from scratch, addressing the above problems. The major change was discarding the bytestream file model and accessing files on a record basis; §3.1 discusses the benefits (and drawbacks) of this. Two further extended operations were also added, namely duplicating records within a file and comparing records between files. It would be nice to report that extended operations were derived from analysis of the behaviour of real programs, but in fact the choice of extended operations is somewhat *ad hoc*. An overall design was drawn up, separating the file access protocol requirements into separate transport and application layers, as shown below.

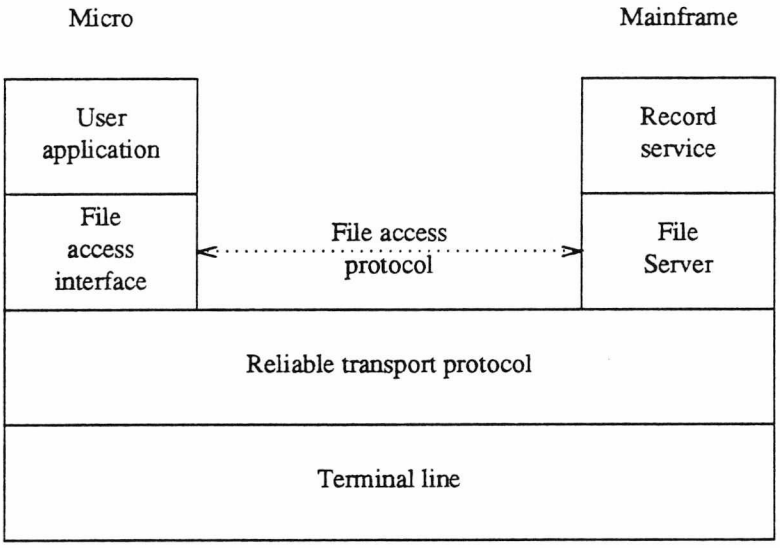

Figure 1.1: File access system overall design

A user application passes requests to the file access interface. This parcels up the request into a file access protocol transaction, which is sent to the server over a reliable transport protocol. The server uses local record-handling services to fulfill the request and returns the result. The user application may abort this process at any time by submitting an *abort operation* request. The application may carry on with other work while monitoring the interface awaiting the arrival of the result. At no time does the application have to surrender control to the library for any significant time.

The modularity of the design allows the transport protocol to be changed independently of the file access protocol and vice-versa. The server may use such local record-handling capabilities as are available and can provide functions not available in the most efficient manner possible for the mainframe system.

The bulk of the rest of this thesis describes the transport protocol (Chapter 2) and the record library and file access protocol and interface (Chapter 3). Two applications using the system are described in Chapter 4, and lastly Chapter 5 gives a summary, an outline of possible future directions and overall conclusions.

Chapter Two

The Transport layer

This chapter covers the provision of a reliable link over a terminal line connecting microcomputer and mainframe. It begins with a short description of some asynchronous serial protocols, mostly file transfer protocols, in use today. It then describes the design and implementation of MMMS, a protocol designed to provide a reliable, two-way inter-application transport service as required by the file access system. The chapter concludes with a short discussion of MMMS performance and some comments on MMMS, including areas for improvement.

2.1. Some current microcomputer asynchronous protocols

This section describes in some detail the inner workings of six asynchronous serial protocols in current use on microcomputers, and concludes by highlighting some other noteworthy protocols. The six protocols described are Xmodem, Kermit, Compuserve B, Zmodem, X.PC and UUCP 'g'. All but the last two are file-transfer protocols; X.PC is a link-level protocol similar in concept to X.25 and the 'g' protocol is used by the UUCP file transfer system to provide a reliable link-level service. They are all in the public domain.

The protocols are all *Automatic Repeat Request (ARQ)* protocols. Data to be transmitted is split into conveniently-sized chunks, or *packets*, and then transmitted with some form of packet identification and a *checksum*†, calculated from the contents of the packet. On arrival at the receiver, the checksum is recomputed from the packet contents and compared to the received value. Any discrepancy indicates a corruption during transmission, and the receiver requests that the packet be resent.

A simple checksum may be calculated by summing the contents of the packet and discarding all but the bottom *n* bits of the result. Some of the above protocols employ just this technique, perhaps with minor variations. Others employ a more effective algorithm based on the use of polynomials to produce a *Cyclic Redundancy Check (CRC)* value [McNamara77]. The use of CRCs greatly reduces the chances of errors

† 'Checksum' is used here and below to mean a check value; it does *not* imply that the value was obtained from an arithmetic sum.

slipping through undetected.

The simplest method of exchanging packets is for the transmitter to send a packet, and then wait for the receiver to reply, either sending a positive acknowledgement (*ACK*) that the packet was received correctly, or a negative acknowledgement (*NAK*) requesting a retransmission. If no answer is received after a specified time, the transmitter *times out* and resends the packet. Because the transmitter must wait for a reply before sending the next packet, the transmission channel lies idle during this time, so this process is known as *Idle-RQ*.

Better use of the transmission channel bandwidth can be made if the transmitter sends packets continuously — *Continuous-RQ*. The receiver sends ACKs or NAKs as packets are received. This is more complex, as the transmitter has to keep a record of which packets have been sent but not yet acknowledged, and if necessary retransmit them. Two strategies for retransmission may be used.

Go-Back-N

When the transmitter receives a NAK for packet n, packet n and all succeeding packets are retransmitted. For example, suppose packets n, $n + 1$, $n + 2$ are transmitted, and then a NAK arrives for packet n. Packet n is then retransmitted, followed again by $n + 1$ and $n + 2$; the receiver will have ignored them previously because they not in the expected sequence.

Selective Retransmission

If the transmitter receives a NAK for packet n, only packet n is resent. The receiver accepts other correct packets despite their being out of sequence.

Selective Retransmission avoids the unnecessary retransmission of packets required by Go-Back-N. On the other hand, it requires more buffer space (packets received out of sequence must be stored), and is more complex to program. In both cases, a limit — the *window* size — is set on the number of packets that may be outstanding (i.e. sent but unacknowledged) at once. Continuous-RQ protocols are sometimes referred to as windowing protocols.

If data can flow in both directions simultaneously, it is possible to *piggyback* acknowledgements for data packets received onto outgoing data packets. Further elaboration on this and the other issues covered in

this section may be found in [Cole82] and [Tanenbaum81].

Because the number of bits in a byte may vary depending on the computer under consideration, *octet* is used below to signify an 8 bit quantity. Note also that the link may restrict communication to 7-bit quantities, possibly using any eighth bit as a parity check, or may assign special meanings to various values (control codes). In this case a distinction is made between octets of file data and *characters* actually transmitted.

All the following protocols use ASCII control characters for one purpose or another. An ASCII table is given in Appendix H.

2.1.1. Xmodem

Xmodem is the simplest protocol of the six described, as well as being the oldest. It requires a completely transparent link — all 256 possible octets must be passed without corruption or otherwise affecting the state of the link. For example, XON/XOFF flow control must be disabled to prevent any XON or XOFF characters in the data from confusing the link.

Xmodem uses four ASCII control characters, SOH, ACK, NAK and EOT.

2.1.1.1. Packet format

SOH
Packet no.
255 — Packet no.
Data octet 1
...
Data octet 128
Checksum

Figure 2.1: Xmodem packet format

Xmodem packets have the format given above. Packet numbers start at 1, increment by 1 and wrap around from 255 to 0. Packets are always 132 octets long (128 data octets plus 4); this is a reflection of Xmodem's origin on CP/M microcomputers of the late 1970s, data in CP/M disc files being held in 128-octet blocks. It also means that up to 127 octets of unwanted data can be added to the end of the file being transferred. The checksum is simply the least significant eight bits of the sum of the data octets.

2.1.1.2. Packet exchanges

Xmodem is an Idle-RQ protocol. The sender transmits a data packet and waits for the receiver to reply with either an ACK character, signaling the packet was received correctly, or a NAK character requesting the packet be retransmitted. When all the file data has been sent, the sender transmits a single EOT character and waits for an answering ACK character. The transfer is then complete.

The receiver times out if it has not received a packet after 10 seconds, assumes the packet has been lost and sends a NAK character requesting retransmission. This mechanism is used to initiate the transfer; the sender waits for the receiver to send a NAK character before it sends the first packet.

Before the receiver sends a NAK character, it must wait for the line to clear. This ensures that it receives the rest of the corrupted packet, and does not generate spurious NAKs in response to packet contents resembling a packet header. The usual procedure is to read characters from the line until one second passes without any fresh character appearing. If 10 NAKs fail to produce a valid packet, the transfer is abandoned.

2.1.1.3. Comments and extensions

Apart from the introduction of unwanted characters at the end of a file, the chief defect of Xmodem is its reliance on single-character control messages (ACK, NAK and EOT). This makes it susceptible to spurious acknowledgements generated by line noise. It provides no facilities for exchange of file attributes such as name, size, creation date etc. and no batch transfer facility which would allow many files to be exchanged in one session. There is also no standard way to indicate what optional protocol features may be available, such as those provided by the derivatives below; the most popular method is to replace the first NAK character sent by the receiver by another single character indicating the receiver capabilities. If the

first packet does not arrive within some short time, the receiver reverts to sending a NAK. In cases where several optional facilities are available, the receiver may offer several capabilities before reverting to standard Xmodem.

Many Xmodem derivatives exist, all correcting some or other perceived defect. Three of the most widespread are Xmodem-CRC which substitutes a more robust 16-bit CRC for the simple checksum, Ymodem which adds a batch transfer facility and optional 1028-octet packets to improve transfer speeds, and WXmodem which converts it to a Continuous-RQ protocol with Go-Back-N retransmission and relaxes the link transparency requirements somewhat by encoding three selected control characters (XON, XOFF and SYN) before sending them.

2.1.2. Kermit

Kermit places few restrictions on the link, having been designed specifically for file transfer to and from mainframes using terminal lines which are frequently anything but transparent. It requires only that the link be able to pass unmolested all printable ASCII characters (space to tilde) and one, selectable, control character. Note that this means it can work over 7-bit links. It achieves this by *quoting*; two printable characters (by default '#' and '&') are set aside as the *control quote* and *eighth-bit quote*. When the data to be sent includes an octet which is a control character, it is sent as the control quote character followed by a printable representation of the control character. For example, Control-A would be encoded as ''#A''. Similarly for octets with the most significant bit set. Some octets require both quotes, in which case the eighth bit quote is sent first.

Kermit uses three character transformation functions. *tochar* transforms integers in the range 0 — 94 by adding 32 to generate a printable ASCII character. *unchar* is the inverse of *tochar*. *ctl* generates a printable character from a control character, and is its own inverse.

Kermit specifies that all communication must take place using ASCII characters. It also distinguishes between binary and text files, and when sending text requires all text lines to be terminated with an ASCII CR-LF sequence.

2.1.2.1. Packet format

The Kermit packet format is as below.

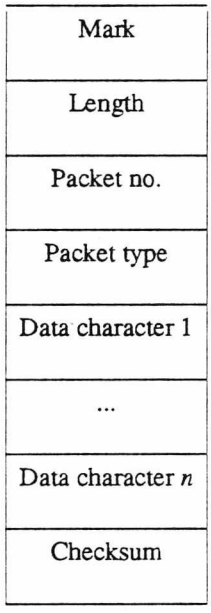

Figure 2.2: Kermit packet format

Mark is the one control character used by Kermit, and indicates the start of a packet. The default Mark character is ASCII SOH. Length is the number of characters following the Length character, up to and including the checksum character, encoded as a printable character using *tochar*. This means that the maximum number of data characters in a packet is 91. Packet numbers run from 0 to 63, and again are sent encoded as printable characters using *tochar*. Response packets from the receiver carry the number of the packet to which they are replying.

The nine basic packet types used by Kermit are listed below. In fact full Kermit defines no less than 17 packet types; the use of the remaining eight is outside the scope of this discussion.

S	Send Initiation
F	File Header
D	File Data
Z	End of file
B	Break (end of transmission)
Y	Acknowledgement (ACK)
N	Negative Acknowledgement (NAK)
E	Fatal Error

Table 2.1: Kermit packet types

Note that acknowledgement of packets is done with a special acknowledgement packet. For the rest of this discussion on Kermit, ACK and NAK refer to the packets of that type, not the ASCII characters.

Kermit allows for several types of checksum, all calculated from all characters in the packet except for the mark and the checksum itself. The default is a simple arithmetic sum, discarding all but the bottom eight bits and combining the two most significant of these into the lower six. This produces a numeric value in the range 0 — 63, which is then encoded using *tochar*. Alternative checksum options include a 12-bit checksum encoded as two printable characters, and a 16-bit CRC encoded as three printable characters.

2.1.2.2. Packet exchange

Kermit is an Idle-RQ protocol. A file transfer proceeds as follows. The sender starts by transmitting a Send Initiation packet. The data field of this packet is filled with various configuration options; the meaning of successive data characters is as follows. Again, an exhaustive list is outside the scope of this discussion; only the minimum options are given. Many of the values are subject to one or other of the standard Kermit transformations — these are indicated.

MAXL	Maximum length of packet to send me. *tochar*
TIME	Timeout. Seconds to wait before timing me out. *tochar*
NPAD	Number of padding character I would like to precede each packet. *tochar*
PADC	The padding character to use, if NPAD is not zero. *ctl*
EOL	Send this character after each packet. Typically CR. *tochar*
QCTL	Use this character as the control quote character.
QBIN	Use this character as the eighth-bit quote character. If the character is 'N' then eighth-bit quoting is not done.
CHKT	The type of checksum to use on packets sent to me.

Table 2.2: Kermit initiation parameters

The receiver responds with an ACK packet, the data field of which is filled with its configuration options. The sender then transmits a File Header packet containing the filename of the file being transferred (converted to upper case with all device names, directory names, version numbers and so on stripped). The receiver acknowledges this with an ACK packet. The sender than transmits successive data packets containing the file data, each of which is acknowledged. When the last data packet has been received properly, the sender transmits an End of file packet. The receiver acknowledges as usual. If there are further files to send, the sender transmits the File Header packet for the next file, and carries on as before, or if the session is complete sends a Break packet and exits without waiting for an acknowledgement.

Both the server and the receiver operate timeouts. If the sender times out before receiving an acknowledgement for a packet, it resends that packet. If the receiver times out before receiving a packet, it re-requests that packet by sending a NAK packet. If no response is forthcoming after a specified number of tries, an error packet is sent with a description of the error in its data field and the transfer is abandoned.

2.1.2.3. Comments and extensions

Kermit's great strength is undoubtedly its portability, in the sense that it is possible to provide a Ker-

mit implementation on virtually any extant computer system. The price paid for this flexibility is in raw transfer speed — the combination of short packets and the necessity for quoting characters means that a Kermit transfer will take longer than with any of the other file transfer protocols reviewed here.

The initial exchange of configurations allows extensions to the basic protocol to be added in an organised manner. Since Kermit's inception, its designers at Columbia University have specified many optional extensions to the basic protocol, outlined below.

— *Run-length encoding*. This allows sequences of the same character to be replaced by a repetition count and one instance of the character. A simple form of data compression, but effective in some circumstances.

— *Attribute packets*. These can be sent after the File Header packet but before the first data packet, and specify any of a large range of file attributes such as file length, creation date, file type, data encoding, machine of origin etc.

— *Server operation*. A Kermit program operating as a server accepts a wide range of commands. File transfers in either direction can be requested; other commands available include commands to change the current server directory, delete/rename/copy etc. files, and list the current server directory. All commands and results are sent in Kermit packets. Using Kermit in this way can avoid a lot of switching back and forth between mainframe and microcomputer environments when transferring multiple files.

— *Sliding windows*. Enabling this option allows Kermit to function as a Continuous-RQ protocol (with selective retransmission), which greatly enhances its performance.

— *Long packets*. Over links with relatively low error rates, Kermit's performance can be enhanced by the use of longer data packets, as this reduces the amount of protocol data that has to be sent for a given amount of file data. This extension allows packets to be up to 9024 characters in length.

— *Extra-long packets*. Some modern modems provide high-speed error-free connections, but only give peak performance when handling data in quantities exceeding even 9024 characters. This extension allows packets to be up to 857,374 characters long.

It should be stressed that these optional extensions are just that — optional. Implementations of the last three are still rare; in particular, the author is not aware of any implementation of extra-long packets.

2.1.3. Compuserve B

The Compuserve B protocol, while requiring a link that will pass eight-bit characters, is slightly less stringent in its requirements than Xmodem. It does cater for links which use XON/XOFF flow control, and links that do not pass NULs reliably. This is no doubt a reflection on the restrictions imposed by the lines into the Compuserve service.

It is also the earliest of the protocols described to have defined a startup sequence by which, given appropriate terminal software†, the mainframe can command the microcomputer to commence a transfer — the whole process can thus be driven solely from the mainframe command line. The startup sequence is as follows— ENQ, DLE, CR and ESC are ASCII control characters, literal characters are enclosed in ''.

Mainframe		Microcomputer
ENQ	→	
	←	DLE 'O'
ESC 'I'	→	
	←	'#XXX99,?,?' CR

Figure 2.3: Compuserve B startup sequence

In the last message, XXX is a 3 character product identifier code; the default for a generic terminal device is 'DTE'. Following it is a version number in decimal; this takes as many digits as necessary, or may be omitted. Some comma-separated feature codes may follow. A disc transfer will typically have feature codes 'PB', meaning the B protocol, and 'DT' requesting a disc transfer. This completes the startup sequence. All data exchanged is now in packets.

As with Kermit, Compuserve B distinguishes between text and binary files and when sending text

† The B protocol is defined in a document that specifies a virtual terminal VIDTEX for use with Compuserve.

requires the end of a line of text to be indicated by an ASCII CR-LF sequence.

2.1.3.1. Packet format

Compuserve B packets are formatted as follows. Note that the packet length is not supplied; instead, a special character delimits the end of the packet. Packets can be up to 512 characters in length.

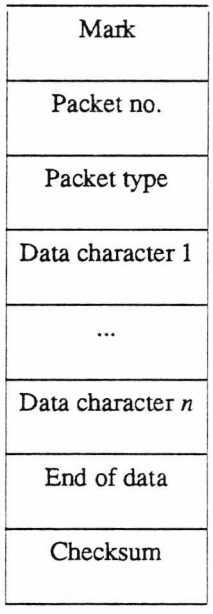

Figure 2.4: Compuserve B packet format

Mark is a two-character sequence, DLE 'B', indicating the start of a packet. The packet sequence number is a single character between '0' and '9' inclusive. The End of data character is ETX. If any of the ASCII characters NUL, ETX, ENQ, DLE, NAK, XON, XOFF occur either as a data octet or in the checksum, they are sent encoded as DLE '@' *c*, the latter character being the result of adding the octet to the character code for '@'.

The checksum is calculated by combining every character into an ongoing total as in the code fragment below. *c* is the character being added in to the checksum *cksum*.

```
cksum := cksum shl 1 ;
if cksum > 255 then
   cksum := ( cksum and 255 ) + 1 ;
cksum := cksum + c ;
if cksum > 255 then
   cksum := ( cksum and 255 ) + 1 ;
```

Compuserve B uses three packet types.

T	Transfer command
N	Data
F	Failure

Table 2.3: Compuserve B packet types

2.1.3.2. Packet exchange

After the startup sequence has completed, the mainframe sends a Transfer command packet. This indicates that either a download from mainframe to microcomputer or an upload from microcomputer to mainframe has been requested. It also indicates whether the file is a text or binary file, and gives the filename. This packet is acknowledged using the standard acknowledge sequence, a DLE character followed by the packet number of the packet in question. Compuserve B is strictly an Idle-RQ protocol.

The sender having been determined, the file data is transmitted in a sequence of Data packets, each of which must be acknowledged by the receiver. After the last data packet has been sent, a Transfer command packet is sent indicating the end of the file. It is acknowledged and the protocol terminates.

In the event of a packet being incorrectly received, a negative acknowledgement is sent in the form of a NAK character (note that it is not accompanied by a packet number). The receiver can time out, abandon the packet and send a NAK if the flow of characters in a packet dries up before the end of the packet is reached. If the sender gets into difficulties, it can send an ENQ character, to which the receiver replies by acknowledging the last packet it received correctly. In the event of an error, such as the file not existing or a user abort, a Failure packet is sent with its first data character being a code indicating the cause of failure.

2.1.3.3. Comments and extensions

While slightly more robust than Xmodem, it is still easy for line noise to generate a spurious negative acknowledgement, or worse a spurious acknowledgement for a packet. The latter is less likely than with Xmodem; the line noise must also generate a valid packet character. The protocol is not widely available

for file transfer other than on the Compuserve service, and it has not attracted the same attention as Xmodem and Kermit.

An extension, the Quick-B protocol, has recently been made public. This includes a capability negotiation transaction, 16-bit CRC packet checksum, maximum packet size extended to 1024 data characters and Continuous-RQ operation with Go-Back-N retransmission. It also extends the quoting mechanism to quote all characters Kermit can quote (i.e. all control and eighth bit characters), though using different quoting characters. However, it retains single-character NAKs and two-character ACKs.

2.1.4. Zmodem

Before outlining the Zmodem protocol, it should be stressed again that Zmodem is **not** a variation on Xmodem. The development of Zmodem did start as a attempt to rectify problems of reliability and throughput with Xmodem, but as the design progressed it became clear that a fresh start was needed.

Zmodem's link requirements fall between Compuserve B and Kermit. It requires an 8-bit link, and by default avoids using the ASCII characters DLE, XON and XOFF. They are sent using the same mechanism Kermit uses, a special character (not the same as Kermit's) followed by a printable representation of the character in question obtained from the *ctl* function. For use over troublesome links, all control characters may be escaped, except one, ZDLE. ZDLE indicates the start of a Zmodem control sequence, and is defined as ASCII CAN. The design permits extension to deal with 7-bit links, but no such extension has yet been specified.

Zmodem addresses problematic areas in various systems' handling of serial I/O. It can be used on microcomputers that cannot overlap serial I/O and disc I/O, microcomputers that cannot overlap serial send and receive and systems which cannot test for the presence of serial input without waiting for input data to arrive. These are important considerations for Zmodem, because it is by design a Continuous-RQ protocol. The Continuous-RQ variants on the protocols above cannot be implemented on systems that suffer from one or more of the above problems. Zmodem also minimises traffic from receiver to sender, to maximise throughput when using some contemporary high-speed modems†.

† These modems, such as the Telebit TrailBlazer, use a high-speed half-duplex link which is imitates a full-duplex link

Again, a distinction between text files and binary files is drawn. Text lines may be ended with one of two terminators — translation can be forced if necessary.

Zmodem defines a rudimentary startup sequence, which allows the sending program to start the receiving program automatically. The startup sequence consists simply of sending "rz" followed by a carriage return; if the sending program is at the microcomputer, this will usually be enough to start the Zmodem receiver from the mainframe command line. Microcomputer terminal programs supporting Zmodem transfers can also monitor the line for the ZDLE character, ASCII CAN. This character is not used in most ASCII terminal operations, and so the following characters can be scanned to determine if they are the start of a packet from a sender or receiver initiating transfer.

2.1.4.1. Frame format

All Zmodem data is exchanged in *frames*. A frame consists of a frame header followed by zero or more data subpackets. A frame header is formatted as follows.

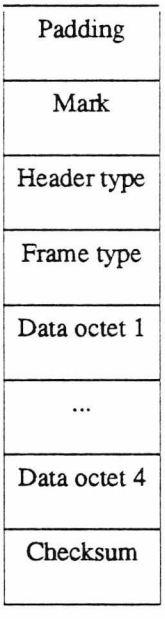

Figure 2.5: Zmodem frame header

All packet contents from Frame type onwards may be control-quoted if necessary. There are three types of

by occasionally reversing the direction of data flow. Reversing the link takes an appreciable time, and hence reduces throughput.

frame header, namely binary with a 16-bit CRC checksum, binary with a 32-bit CRC checksum and a printable hexadecimal encoding with a 16-bit CRC checksum. Each character in this latter type from Frame type onwards is sent as a pair of hexadecimal digits. The receiver always replies using these hex frames.

The Zmodem Mark character is ZDLE. It is always preceded by a Zmodem padding character ZPAD ('*'); hex frames are preceded by two padding characters. Hex frames are also always followed by ASCII CR-LF. These aid debugging from traffic printouts, and also overcome problems with some operating systems.

There are 19 different Frame types in all. The following table lists the 8 main ones.

ZRQINIT	Request receive init
ZRINIT	Receive init
ZACK	Acknowledge data received
ZFILE	File name from sender
ZFIN	Finish session
ZRPOS	Resume data transmission here
ZDATA	Data subpacket(s) follow
ZEOF	End of file

Table 2.4: Zmodem frame types

Only some of the above make use of the four octets of data available in the frame header. ZRINIT and ZFILE use them to pass configuration information and transfer options respectively. ZACK and ZRPOS pass the byte offset in the file from which the receiver is expecting to receive data; ZACK merely confirms correct reception up to that point, ZRPOS demands retransmission from that point. ZEOF contains the number of file octets that have been transferred, i.e. the file length. ZDATA contains the file offset of the data following in subpackets.

Frame headers may be followed by data subpackets. These are formatted as below.

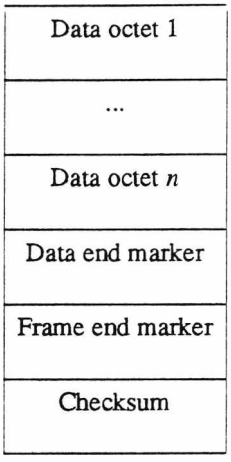

Figure 2.6: Zmodem data subpacket format

The checksum is calculated over all the packet contents other than the checksum itself. All data octets may be control-quoted. The Data end marker is ZDLE. There are four different Frame end markers, indicating whether this is the last data subpacket in the frame or not, and whether an acknowledgement to this sub-packet should be sent. Currently only binary subpackets are specified; the standard makes provision for ASCII encoded data subpackets which would allow use over 7-bit links. Recommended subpacket lengths are 256 octets below 2400 bits per second (bps), 512 at 2400 bps and 1024 above 4800 bps or when the link is known to be relatively error-free.

2.1.4.2. Packet exchange

All Zmodem exchanges begin with the receipt by the sender of a ZRINIT frame. The receiver sends this frame when it starts; the sender may request it be resent by sending a ZRQINIT frame. The sender then transmits a ZFILE packet followed by a data subpacket giving the file name and other file information. The receiver replies with a ZRPOS frame indicating the file offset from which transmission will start. This will usually be 0, but other values are possible, for instance if the transfer is being restarted after a system crash. The file data is then exchanged by one of a variety of methods outlined below, and a ZEOF frame sent to indicate the end of the file. The receiver responds with a ZRINIT frame, at which point another file transfer can commence. If there are no more transfers to be done, the sender transmits a ZFIN frame, to which the receiver responds with another ZFIN frame. Finally the sender transmits 'OO' (Over and Out) and exits; the receiver waits briefly for these characters and exits regardless.

Five strategies are available for the exchange of file data, requiring different levels of receiver capability.

1. *Full Streaming with Sampling*. This can only be used if the receiver can overlap serial I/O with disc I/O, and can scan the reverse channel for the presence of data without having to wait. Data transmission starts with a ZDATA header and continuous subpackets, none of which require acknowledgement. If a subpacket becomes lost or garbled, the receiver sends a ZRPOS frame giving the file offset marking the end of the last correctly-received file data, and ignores all incoming frames until a ZDATA frame retransmitting from that offset arrives. The sender must check the reverse channel for a ZRPOS frame before transmitting each data subpacket.

   The sender can monitor the progress of the receiver by requesting that subpackets generate ZACK responses. It can then avoid 'getting ahead' of the receiver to a degree that would slow error recovery unduly.

2. *Full Streaming with Reverse Interrupt*. If the sender cannot examine the reverse channel without having to wait for some data to arrive, it may be possible to send a sequence that interrupts the sending program. This sequence, known as the *Attn* sequence, may be specified by the sender using an optional frame carrying sender parameters. When an error is detected, the receiver sends the Attn sequence (for a Unix system this will normally be the interrupt character — e.g. Control-C — and a short delay to give the sender time to respond) followed by a ZRPOS frame. Recovery then proceeds as above.

3. *Full Streaming with Sliding Windows*. If neither of the above methods are possible, but the sender can buffer responses from the receiver, the sender can request that all data subpackets are acknowledged with ZACK frames. After a sufficient number of subpackets have been sent, the sender can expect to find a ZACK frame waiting in its receive buffer.

4. *Full Streaming over Error-Free channels*. If an error-free channel is available, the entire file may be sent in one data subpacket. There is no need to check the reverse channel while sending.

5. *Segmented streaming*. If the receiver cannot overlap serial I/O and disc I/O, it may specify a buffer length which the sender must not overflow. The sending program then sends data subpackets

requiring acknowledgement, and waits until the replying ZACK arrives before sending the next sub-packet.

The timing of transfers is controlled by the receiver. The sender does not time out at all (except, perhaps, to abort if it is expecting traffic and none has been received after a significant period). If the receive program does not receive an expected frame, it times out after about 10 seconds and retransmits its last frame.

2.1.4.3. Comments and extensions

The last section outlined the basic operation of Zmodem. The specification allows for many facilities not mentioned. The more interesting of these are outlined below. It should be noted that many of these, although provided for, are awaiting detailed specification.

Escaping octets with the 8th bit set. This would allow Zmodem to be used over 7-bit links.

Conversion of end of line convention. When sending or receiving text files, Zmodem can be requested to convert local end of line conventions to or from one of the two supported end of line conventions recognised by the protocol.

Compare files before transferring. If a filename matching the one of the file to be transferred already exists on the receiving system, its length and a CRC of its contents may be returned to the sender. If these match the file to be sent, the transfer is assumed to be unnecessary and abandoned.

File management options. The protocol allows the sender to instruct the receiver on various file management matters concerned with the transfer. Possible options include not overwriting an existing file of the same name, updating an existing file or appending to an existing file.

Compression and encryption. The file data sent to the receiver may be compressed using run-length encoding or Lempel-Ziv compression with 12-bit encoding [Welch84]. Alternatively the data can be encrypted.

Sparse files. Sparse files may be exchanged (assuming the facility is supported by both sides) by sending each segment as a separate data frame, and requesting that the last data subpacket return an acknowledgement. The sender waits for that acknowledgement to arrive before proceeding with the

next segment.

Command execution. The sender can indicate that the data following is not file data, but a command to be executed either by the receiving application program or by the receiving operating system command interpreter.

Zmodem answers all of the criticisms levelled at the protocols outlined above. Its chief virtues are that it is robust and makes efficient use of available bandwidth. The protocol document reports observed efficiencies of 95% (i.e. observed throughput of *file* data octets of 915 octets per second on a 9600 bits per second line).

The price of this improvement is the additional complexity of Zmodem over the other protocols reported. As a consequence, and also due to its relative youth (at the time of writing, Zmodem is under two years old, while Kermit has been widely available for at least four years), implementations are not as widely available as for Xmodem and Kermit. However, public domain C code implementing the protocol has been made available, and the performance gains even over Continuous-RQ versions of Xmodem and Kermit will undoubtedly ensure Zmodem's popularity.

2.1.5. X.PC

X.PC [Tymnet83], as mentioned at the start of §2.1, is a link-level protocol rather than a file transfer protocol. It provides a transparent error-free link; to transfer files a separate file transfer protocol running on top of X.PC may be used. One example of such a protocol is FAST [Nixon86]; Zmodem with full streaming can also be used. It is included here as an example of an asynchronous link-level protocol aimed at microcomputers. Another such protocol is MNP which is widely supported by high-speed modems in the USA. Unfortunately, the design of MNP is proprietary to its developers, Microcom Inc., who license it to other modem manufacturers. X.PC is in the public domain.

X.PC was designed by Tymshare Inc. in 1983 to facilitate use of microcomputers with their X.25 network, Tymnet. X.PC is based on X.25 [CCITT84] and provides nearly all X.25 facilities. The description of the protocol given below confines itself to an outline of the packet and frame structures and highlighting differences between X.PC and X.25. The reader is assumed to be familiar with X.25; a full description is

not appropriate here.

X.PC was designed for use with modems connected to Tymnet via a Tymnet X.PC network server. It therefore requires a transparent full-duplex 8-bit line. X.PC allows up to 15 logical channels to be multiplexed onto one line. Unlike the file transfer protocols above, data may be exchanged in both directions simultaneously.

2.1.5.1. Packet and frame formats

X.PC exchanges data in packets. The format of a packet is as follows.

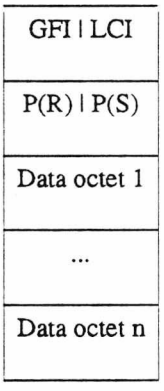

Figure 2.7: X.PC packet format

GFI (General Format Identifier) is held in bits 4-8† of the first packet octet. Bit 8 identifies the packet as either a data packet or control packet. If the packet is a data packet, bits 5-7 hold various control flags‡. LCI (Logical Channel Identifier), held in bits 1-4 of the first packet octet, indicates the logical channel through which the packet is being sent. P(R) and P(S), held in bits 5-8 and 1-4 of the second packet octet, are the sequence numbers of the most recent packet to be received by the sender and this packet respectively. If the packet is a control packet, Data octet 1 holds the packet type identifier (control packets are as X.25); if the packet is a data packet it holds an octet of user data.

Packets are sent enclosed in frames, one frame per packet. The frame serves to delimit packets and adds error-detection. The format of a frame is given below. Packet data is indicated in *italics*.

† Bit 1 is the least significant bit.

‡ The D, Q and M flags. The use of these is not explained here.

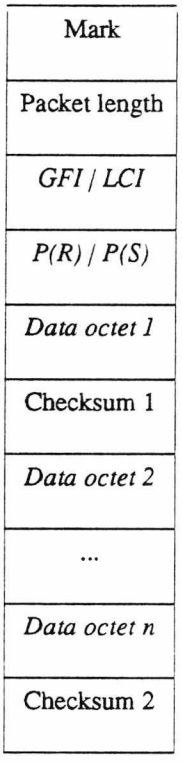

Figure 2.8: X.PC frame format

The X.PC Mark character is ASCII STX. This marks the start of a frame. Packet length gives the number of Data octets following Checksum 1; it does not include the GFI/LCI and P(R)/P(S) octets or Data octet 1. The maximum number of data octets in a packet is therefore 256, while the default is 128. If the packet length is zero, the frame ends after Checksum 1. Checksums are calculated using the CCITT standard 16-bit CRC.

2.1.5.2. Packet exchange

X.PC is a Continuous-RQ protocol with Go-Back-N retransmission. Once a call is established, data is sent in data packets which are acknowledged either by data packets flowing in the other direction or by Receive Ready (RR) packets. Lost or corrupted packets are discarded and their retransmission is requested by Reject (REJ) packets. Flow control is performed using Receive Not Ready (RNR) packets. Packet traffic is exactly as in X.25.

2.1.5.3. Comments and extensions

The X.PC packet differs from the X.25 packet in several minor ways. Since only 15 logical channels are supported by X.PC, the Logical Channel Identifier is stored where X.25 holds the Logical Channel Group Number, and the need for a separate octet for the Logical Channel Number is eliminated. Some X.PC control packets, such as those handling call request, acceptance and clearing, require sequence numbers, so all control packets carry an octet devoted to these; with rearrangement of the GFI this allows packet sequence numbers to range from 0 to 15 rather than 0 to 7 as in X.25.

X.PC offers an optional reconnection facility, which allows virtual calls to be maintained if the physical connection is lost. If this facility is used, reconnect keys are exchanged when the call is requested. Should the connection be lost, then when the connection is re-established a call request is issued on the same logical channel accompanied by both reconnect keys. RR packets are then exchanged to indicate the next packets expected by both sides, and data exchange continues.

2.1.6. UUCP 'g'

The Unix *uucp* (*U*nix-*U*nix *C*o*P*y) command forms the basis of the worldwide UUCP mail network†. It schedules the transfer of requested files between neighbouring participating Unix systems. In the majority of cases the connection is a dial-up asynchronous serial line. Once connection is established, an initial exchange indicates the work to be done and selects a protocol to be used to ensure reliable transfer of data — the process is described in [Nowitz82]. The 'g' protocol is invariably used over dial-up lines, and indeed is the only protocol supported by many UUCP hosts. This means that the 'g' protocol is amongst the most widely-used asynchronous serial protocols. Although traditionally used between mainframe UUCP hosts, it is beginning to spread onto microcomputers.

A brief overview of the 'g' protocol is given below; a full description may be found in [Chesson88]. It should be noted that the protocol is used only for data transfer — all file transfer information, such as file name and direction of transfer is exchanged before the protocol is started, and the protocol is closed down

† The UUCP network is among the networks described in [Quarterman86].

after all file data has been transmitted — and so should be regarded as a link-level protocol. It requires a completely transparent full-duplex 8-bit link.

2.1.6.1. Packet and frame formats

A 'g' packet consists of one control octet followed by a data segment of zero or more data octets. The format of the control octet is as follows.

bit	7	6	5	4	3	2	1	0
	t	*t*	*y*	*y*	*y*	*x*	*x*	*x*

Figure 2.9: UUCP 'g' control octet format

The *tt* bits determine the type of the packet. Packet types are:

00	Control packet
10	Data packet
11	'Short' data packet

Table 2.5: UUCP 'g' packet types

Non-control packets are followed by a data segment. Data packets have a maximum-size data segment, while 'short' data packets indicate the difference between maximum size and actual size at the start of the data segment. The maximum data segment size is determined during initialisation.

In a non-control packet, the sequence number of the packet is given by *xxx*, and the sequence number of the last correctly-received packet is in *yyy*.

There are 6 control packets, numbered as follows with *xxx* indicating the control packet type and *yyy* carrying an accompanying value. If several control packets are to be sent, the lowest numbered is sent first.

xxx	Name	*yyy*
1	CLOSE	
2	RJ	Sequence number to retransmit from
4	RR	Last correctly-received sequence number
5	INITC	Window size
6	INITB	Maximum data segment size
7	INITA	Window size

Table 2.6: UUCP 'g' control packets

The window size is in the range 1-7 (3 is the usual value). The maximum data segment size is 32×2^{yyy}, giving a range from 32 to 4096 octets.

Packets are sent encoded into frames, with one frame per packet. As with X.PC, the frame delimits packets and adds error-detection information.

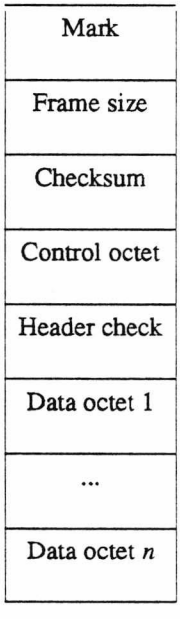

Figure 2.10: UUCP 'g' frame format

Mark is the ASCII DLE character. The frame size *f* must be in the range 1 to 9; 9 indicates a control packet, while other values indicate that the number of data octets in the frame is $32 \times 2^{f-1}$. Hence data frames are

of fixed sizes. When sending 'short' data packets extra octets must be sent to fill the frame. This creates some overhead, but has the advantage that I/O devices can be programmed to take advantage of the constant data frame size.

The checksum is a 16-bit number. If the packet is a data packet, the checksum is calculated from data values using a similar algorithm to Compuserve B. If a control packet, the checksum value is the control byte. Before sending, the checksum is subtracted from 43690 (hexadecimal AAAA), and the result sent low-octet first. Finally, the header check is formed by exclusive-ORing the contents of the frame size, checksum and control octet fields of the frame.

2.1.6.2. Packet exchange

The protocol is initialised by both sides repeatedly sending INITA control packets until an INITA packet is received. An INITB is then sent in reply, followed by an INITC packet when an INITB packet is received. When an INITC packet is received, the channel is considered open, and any INIT packets received henceforth are ignored.

The 'g' protocol is a Continuous-RQ protocol with Go-back-N retransmission. Acknowledgements for data packets received are either piggybacked onto outgoing data packets, or sent in RR packets. The receiver may choose to ignore all errors, in which case the transmitter must time out and retransmit, or may send RJ packets requesting retransmission from a lost or corrupt packet.

The CLOSE control packet is used to terminate communication. When one end wants to close down, it sends CLOSE packets until either one is received or a programmable limit is reached. Receipt of a CLOSE packet causes a CLOSE packet to be returned.

2.1.6.3. Comments

The UUCP 'g' protocol was first distributed with the 7th edition Unix system in 1978, and so is certainly the oldest of the Continuous-RQ protocols described in this section. It has remained unchanged since†, and has started to spread to microcomputers as implementations of the UUCP system such as the

† To be strictly accurate, initial versions had a selective retransmission facility that quickly fell into disuse and is today a historical curiosity.

public-domain UUPC package are made available. It has survived unchanged because it is efficient in use ([Nowitz82] reports observed efficiencies in the region of 90% for lightly-loaded processors), offering little incentive to switch to other protocols — Zmodem may offer better efficiency, but not by much.

2.1.7. Other noteworthy protocols

With the increasing computing power of microcomputers, and the availability of quality high-level language compilers for them, several protocols previously restricted to mainframe computing have begun to appear on microcomputers. The UUPC implementation of UUCP has already been mentioned (§2.1.6.4). A public-domain package written by and for use by radio amateurs exchanges Internet Protocol (IP) datagrams over asynchronous lines using the Serial Line IP (SLIP) protocol; the package provides ICMP, TCP and UDP protocols, and the Internet application protocols FTP (file transfer), Telnet (terminal connection) and SMTP (electronic mail).

Most of the protocols described above assume a transparent or nearly-transparent 8-bit link. One protocol which, like Kermit, can function in more hostile conditions is Blast, a protocol proprietary to Communication Research Group (CRG). Some details of Blast have been made public [Roussel87]; it is a Continuous-RQ protocol with selective retransmission and allows files to be transferred in both directions simultaneously. All packet data is encoded such that packets consist only of upper case and lower case letters, digits and single and double quotes; this allows for use on 7-bit lines and systems which do not pass control characters and some punctuation characters. Packets may be of any length a given implementation will permit, but default to 100 characters. Blast goes further than Kermit in making provision for troublesome connections and systems. However, Kermit is in the public domain and widely implemented, and as those implementations are almost invariably free it is hard to see Blast enjoying anything other than limited success.

2.2. MMMS — Introduction and overview

§1.5 called for a reliable transport protocol capable of exchanging data in both directions between a microcomputer and a mainframe. Of the protocols outlined above, X.PC would be suitable, but suffers from

some drawbacks. Most importantly, its requirement for a completely transparent 8-bit line makes its use over many terminal lines impossible. Kermit, on the other hand, has been implemented on a wide range of systems, communicating via ordinary terminal lines. It is, however, unidirectional, dedicated to file transfer, and not very efficient. It was decided to design a new protocol which would use those features of Kermit that makes it widely implementable and which would also provide the required service efficiently. The resulting protocol was seen as providing a reliable transport service between applications — it was not and is not intended to be a full-blown networking service.

The remainder of this chapter describes the resulting protocol, **M**icro-**M**ainframe **M**essage **S**ervice or MMMS†. MMMS allows data in units of *messages* (a message is up to 1024 octets) to be exchanged reliably and in sequence over full-duplex asynchronous serial lines. Its intended environment is communication between a microcomputer connected to a mainframe via a standard terminal line, or two microcomputers connected through a serial link (perhaps using modems over the public telephone network). The design makes some effort to ensure MMMS will be usable over a wide range of terminal lines.

MMMS fits into the ISO OSI Reference Model as follows‡

OSI	MMMS
Transport	Assures message sequence and integrity.
Network	Not applicable as MMMS is point-to-point.
Datalink	Splits messages into packets, detects errors etc.
Physical	Serial line, system flow control.

Figure 2.11: MMMS and the ISO Reference Model

2.2.1. Portability

MMMS, as mentioned above, is intended for use over terminal lines connected to a mainframe. Kermit has had considerable success in this area, and so heavily influenced the design of MMMS.

† To be pronounced 'mess'.

‡ It should be noted that other interpretations are possible. For example, compare the layering for Kermit given in [Jesty85] to that in [da Cruz87].

The communication issues addressed by Kermit are discussed in [da Cruz87]. It should be borne in mind that Kermit is a file transfer protocol, and so addresses problems that are not relevant to MMMS, such as file attributes and data conversions (e.g. standard form for end-of-line and end-of-data).

MMMS places the following requirements on the communications link:

— communication takes place by means of ASCII characters. The ASCII character set requires 7 data bits to be passed for each character; if the link is capable of passing an eighth bit, and it is not being used for parity, then MMMS is capable of making use of it, but does not require it.

— the link must be full-duplex (i.e. able to send and receive simultaneously), and echoing of received characters must be disabled.

— the link must be able to send and receive the printable ASCII character set (all non-control characters) and one control character (which one does not matter) without interference, be it translation of characters or use of them to invoke control functions.

— the link must either support some form of flow control such as XON/XOFF, or possess an input buffer capable of storing up to 700 characters. In practice, the size of the input buffer required depends on the implementation; 700 characters is an upper bound†.

MMMS transfers data in octets, so will be of limited usefulness to computers which do not deal in word lengths that are multiples of 8 bits.

2.2.2. The protocol

MMMS sends messages by splitting them into packets and exchanging these. Any non-printable message data is encoded printably; in the worst case this can triple the amount of data to be sent. MMMS also carries out run-length encoding [Sedgewick83] on message data before transmission; simply, this means that should the data contain an octet repeated several times, it is transmitted as a repeat count followed by the representation of the octet itself. §2.3 contains further details.

† Up to 7 packets may be transmitted before the sender must wait for an acknowledgement — this is around 700 characters.

Each packet contains up to 94 characters of packet data plus 5 additional characters containing protocol information (packet length, sequence number, integrity check etc.). While Kermit is an Idle-RQ protocol, MMMS is a Continuous-RQ protocol with Go-Back-N retransmission. Selective retransmission was considered but rejected because it was not felt the additional complexity was justified in this case. MMMS may also send data in both directions simultaneously, by 'piggybacking' acknowledgements for packets received onto the data packets being sent. Details of how acknowledgement etc. is accomplished are in §2.3.

2.3. MMMS — Protocol Description

MMMS messages are transmitted by splitting them up into smaller packets which are then exchanged. Packets are numbered to ensure they are exchanged in the correct order, and are transmitted with a checksum to ensure they are received uncorrupted. Internally to MMMS, messages are also numbered to ensure that they arrive in the correct sequence.

2.3.1. Packet structure

A MMMS packet is composed of a succession of ASCII characters, all bar the first printable, that is within the range space to tilde. The packet may optionally be preceded by several padding characters and/or followed by an end-of-line character.

Within the packet, three standard character encoding functions borrowed from Kermit are used. These are **ToChar**, **Unchar** and **Ctl**, and are defined as follows.

ToChar(x) Returns a printable character corresponding to the number x, where x is between 0 and 94 inclusive. The character has the ASCII code obtained by adding 32 to x.

UnChar(x) The inverse of ToChar.

Ctl(x) Returns a printable representation of an ASCII control character, obtained by exclusive-ORing x with 64. For example, Ctl (Control-A) is 'A'. Ctl is its own inverse.

A packet has this structure.

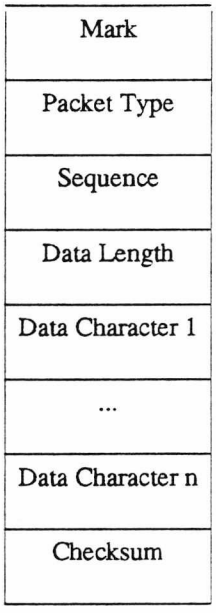

Figure 2.12: MMMS packet format

Mark indicates the start of a packet. It must be an ASCII control character capable of passing uncorrupted through the link, and is the only non-printable character used in MMMS. The default for Mark is ASCII SOH (Control-A).

Packet Type is a character indicating the packet function. Valid packet types are M m L l A a E N S C T and R.

Sequence contains the packet number (if any) and the number of the next packet MMMS expects to receive. Packets are numbered 0 to 7, wrapping round, so a packet number uses 3 bits. The sequence character is formed by taking the packet number XXX (if the packet is not numbered this will be 0) and the number of the next expected incoming packet YYY, combining them to produce a six bit number (msb) XXXYYY (lsb) and converting that value to a printable character with ToChar.

Data Length contains the number of characters of data in the packet, converted to a printable character with ToChar; hence the number of data characters must be in the range 0 to 94. If this value is 0 the packet is empty.

Data Character 1 .. n is/are the data contents of the packet, composed entirely of printable characters. If the packet is empty then there are no data characters.

Checksum is a single character checksum formed from every character in the packet except Mark. Packets received with an incorrect checksum are discarded. The checksum is calculated in the following way (this is the same as the Kermit default single-character checksum):

(i) Calculate the arithmetic sum of all the characters in the packet between (and including) the Packet Type character and the last data character. Discard all but the least-significant 8 bits.

(ii) Add the value of the top two bits (after shifting them rightwards six places) to the bottom six bits, and discard all but the bottom six bits of this result.

(iii) Convert to a printable character with ToChar.

The following code fragment may make the above clearer. 's' is the sum of all characters in the packet, and 'c' the packet checksum character.

```
cksum := (((s and 192) shr 6) + s) and 63 ;
c := ToChar (cksum) ;
```

2.3.2. Packet types

Packets are grouped into three categories, data packets, out of band packets and control packets. Data packets carry both a packet number and the number of the next packet the sender expects to receive — this is used as an acknowledgement for all previous packets. The data packet types are:

M The packet contains message data. The first character of the first data packet for a message is a sequence character. Sequence characters cycle from 'A' to 'Z' and are used to ensure messages are not duplicated.

m As packet type 'M', acknowledging receipt of a NAK.

L The packet contains message data, and is the last packet with data for that message (i.e. the contents of this packet complete the message). Once a message has been received, its sequence letter is examined. If it is the same as one of the previous seven messages received, then a link reset took place between the final packet of that message being received and the acknowledge-

ment reaching the sender; the message has been resent and so can be discarded. If the sequence letter is the one expected the message is made available for collection. Otherwise the connection has become irreparable, and so is abandoned.

l As packet type 'L', acknowledging receipt of a NAK.

A The message currently being sent has been aborted. Discard any data received.

a As packet type 'A', acknowledging receipt of a NAK.

Out of band packets do not require acknowledgement, and so carry no packet number. They do, however, carry an acknowledgement. The out of band packet types are:

E An ACK packet. This is just used to carry the acknowledgement of packets received if no other packets are to be sent.

N A NAK packet. This indicates that the receiver has received packets out of sequence, and requires the sender to resend packets from the packet number given (prior packets can still be regarded as acknowledged). The packet type of the first of these resent packets must be altered to the lower case equivalent, to confirm the reception of the NAK packet. A NAK is also used to restart data transmission halted with a 'S' packet.

S Stop data transmission. The sender of the 'S' packet has run out of buffer space. Data packets following the last one acknowledged as received will be discarded (though the acknowledgement they carry will be honoured). A NAK will be sent when buffer space is available.

Command packets carry neither packet number or acknowledgement. They either have a special acknowledgement or require no acknowledgement. Command packet types are

T Reset link. Any messages being sent, and any sent but with their final packet not yet acknowledged are to be resent from scratch. Any messages being received are discarded. The data field of a 'T' packet contains various host parameters. See §2.3.5 for further details.

R A 'R' packet is sent to acknowledge receipt of a 'T' packet. As with the 'T' packet the data field contains various host parameters.

C A host disconnecting from the link sends a 'C' packet to indicate it is disconnecting; see §2.3.6 for further details.

2.3.3. Data encoding

Individual octets of data within a message may have any value; therefore to send them in a MMMS packet it is necessary to encode unprintable characters in some way. MMMS adopts Kermit's solution to this problem; various prefixes may be placed before a character to indicate transformations that have been applied to the character. There are three prefixes; control prefix, eighth bit prefix and repeat prefix. The characters used for all three may be any printable ASCII character, but should be uncommon. Analysis of a body of English text (volumes 1 and 2 of the Unix manual pages) showed the Kermit standard prefix characters '#' (control), '&' (eighth bit) and '~' (repeat) together comprise 0.03% of the characters used. Analysis of VAX (Unix) and MS-DOS binaries revealed a similarly low occurrence for VAX binaries, but '~' forming about 1% of MS-DOS binaries. Accordingly, '%' is used as a repeat count by the MS-DOS implementation, but the Kermit standard characters are the MMMS defaults.

Terminal serial lines frequently use only 7 bits to transmit a character, perhaps accompanied by a parity bit. If MMMS has an octet to transmit with its top bit set, and the communication link only supports 7 bit transmission, the eighth bit prefix is sent and only the bottom 7 bits of the octet considered further.

If the octet to be sent forms a control character (has a value of 0 to 31), MMMS sends the control prefix and transforms the octet to a printable character with the Ctl function. This printable character is then sent. The control prefix is also used when sending characters that serve as prefixes, to indicate that they are to be treated as normal characters.

Kermit optionally allows characters to be preceded by a repeat count. This consists of the repeat prefix followed by a number between 0 and 94 encoded with ToChar. Receipt of a repeat count indicates that the following character (which may be encoded as above) is repeated by the number of times given. Analysis of the same body of English text as above showed that on average a saving of 12% in characters to be transmitted is achieved. Analysis of Unix VAX binaries and MS-DOS binaries showed savings here to be less uniform (savings tended to be either substantial or almost non-existent), but overall still significant

with an average of 12%. In no case did the presence of an extra prefix character to quote increase the amount of data to be transmitted by 1% or more. Accordingly, MMMS employs the Kermit repeat count prefixing scheme, with a minor improvement. Since repeat counts of 0 and 1 are unnecessary, 2 is subtracted from the repeat count before encoding to a printable character; hence the range of values the repeat count may take is 2 to 96.

Quoting any one octet is done in a fixed order. First a repeat count if any, then the eighth bit prefix if necessary and finally the control prefix if necessary. A few examples of character encoding using the default prefixes are given below.

Octet value(s)	Is encoded as
65	A
01	#A
129	&#A
193	&A
65 65 65	~!A
129 129 129 129	~"&#A
35	##

Table 2.7: Sample MMMS octet encodings

2.3.4. Timeouts

MMMS operates three timeouts, T_1, T_2 and T_3. Timeouts are specified in seconds. MMMS hosts exchange the values of T_1 and T_2 they require during initialisation, and T_3 is determined directly from T_2. All timeouts have default values that apply during initialisation.

If a host does not receive a reply to a packet requiring one before T_1 seconds have elapsed, the packet is deemed lost and is retransmitted. A packet may be retransmitted up to MAX_RETRIES times, after which the link attempts to reset itself (see below). Should this happen to an initialisation packet the link is deemed to be disconnected. MAX_RETRIES is currently defined as 6. The default time for T_1 is 10

seconds.

T_2 is the longest time that should elapse without the host transmitting any packets. Should this timeout fire then an 'E' packet is transmitted acknowledging all packets received so far. The intention here is to keep an idle link ticking over, as many systems will automatically log out a terminal line idle for more than some fixed time as a security measure. The default value for T_2 is 30 seconds.

T_3 is the longest time a host should go without receiving any packets. If this timeout fires the link is considered to be disconnected. The value of T_3 is fixed at $T_2 \times$ MAX_RETRIES, and so has a default value of 3 minutes.

Whatever actual timeout values are used, T_1 must be less than T_2, which must in turn be less than T_3.

2.3.5. Link initialisation

On being requested to establish a connection, MMMS sends a 'T' packet and waits for a response. If no correct response is forthcoming the 'T' packet is repeated up to MAX_RETRIES times or until T_3 fires, after which MMMS gives up and reports the connection could not be established.

In happier circumstances a reply of either a 'T' packet or a 'R' packet will be received. In either case the other host's configuration information is extracted from the packet data field and the link considered established. Should the packet received be a 'T' packet, a 'R' packet is sent in reply with the host's configuration information. Exactly the same thing happens if the link is already established, plus also any messages being sent or received are abandoned, and those being sent or awaiting acknowledgement of their final data packet are put at the top of the queue of messages to send to ensure they are resent in the correct order.

The configuration information exchanged by the hosts is a sequence of printable characters in the following format, leftmost first:

M	T	N	P	E	Q	Q	Q	M	M	T
A	I	P	A	O	C	B	R	A	A	I
X	M	A	D	L	T	I	E	X	R	M
L	E	D	C	C	L	N	P	R	K	2

Figure 2.13: MMMS initialisation packet fields

The interpretation of these fields is as follows. 'I' and 'me' refer to the host sending the packet. Numeric values are converted to printable characters using the ToChar function, and control characters are made printable using the Ctl function.

MAXL is the maximum number of data characters I want to receive within a packet (in the range 20 to 94). Some hosts may lack the buffer space to hold a full sized packet, whilst others may prefer a small packet size to minimise the effects of a noisy communications line. Hosts should compare maximum packet sizes requested by the other host to their own maximum packet size, and use the smallest; otherwise a two-way transfer may be needlessly held up because the larger packets arrive less frequently and thus make y allow fewer opportunities for acknowledgements for the smaller packets, impeding the flow of the smaller packets. A host may unilaterally adjust the maximum size of packets it sends downwards to suit prevailing conditions, but should not increase packet size above this level. The default for MAXL is 94.

TIME is the number of seconds (1 to 94) to wait for an acknowledgement before retransmitting a packet to me (i.e. the value of T_1 you should use). As with MAXL, hosts should compare the value of TIME requested by the other host to their own and use the longest value; otherwise a mainframe host with a high timeout value to account for delays in its I/O subsystems may time out a more responsive micro while the packet in question is still bogged down within the mainframe I/O subsystem.

NPAD is the number of padding characters with which to precede each packet sent. The default is 0.

PADC is the padding character to use. This is assumed to be a control character, and the default is ASCII NUL.

EOLC is an end of line character to be sent after the last character of every packet. Again, this is

assumed to be a control character. If EOLC is ASCII NUL, no end of line character is sent. The default EOLC is a carriage return, ASCII CR.

QCTL is the character I wish to be sent as the control prefix character. The default is '#'.

QBIN is the character I wish to be sent as the eighth bit prefix character. If this is the same as QCTL, then I believe the link to be capable of transmitting eighth bits without interference. If only one host believes this, and receives a 'R' packet from the other host indicating it can only handle seven bit wide characters, the first host will use the other's eighth bit prefix as its own eighth bit prefix (it cannot use its normal eighth bit prefix as it has no means of informing the other host what this is). The default is '&'.

QREP is the character I wish to be sent as the repeat count prefix. The default is '~'.

MAXR is the maximum number of times you should allow the link to be reset before deciding it is beyond repair and disconnecting. The link may go through the reset procedure (i.e. exchange of 'T' and 'R' packets) during data transfer should the hosts become unsynchronised (e.g. if one host should receive an acknowledgement for a packet it hasn't sent). The default is 10 times.

MARK is the control character I wish to be sent to indicate the start of each packet. Note that if both sides decide to use Mark characters other than the default, then the 'T' and 'R' packets will not be recognised by either side as they will not be preceded by the default Mark character.

TIM2 is the maximum amount of time in seconds (5 to 94) that I wish you to go without sending me a packet (i.e. the value you are to use as T_2). The default is 15 seconds.

Before MMMS can even hope to establish communication, there are many configuration issues that must be resolved and are outside MMMS's control. Both sides must be using the same line speed, for example, and the same parity and word length settings. The default settings for Mark, padding characters and EOLC should ensure most systems can receive the initialisation packets, but it is the nature of things that this may not always be the case. In (hopefully) rare cases (e.g. if neither side can accept the default Mark character) manual intervention may be necessary before everything will flow smoothly.

2.3.6. Link closedown

When a host has finished with a link it should close it down as a preliminary to exiting from the application using MMMS back to ordinary terminal usage. Fatal errors encountered within MMMS also cause the link to close down if messages are currently being passed.

To force a link to close, the host initiating the closedown sends a 'C' packet and marks the link as down. A host receiving a 'C' packet notes the link is down, and abandons any messages being sent or received.

No form of acknowledgement of the 'C' packet is sent or expected. The reason for this lies in the way MMMS is seen as being used. Interaction with the other host will take place as normal using a terminal line. When a MMMS based application is used, communication between hosts is handled by MMMS until the application finishes, in which case use reverts to that of a normal terminal line. Sending just a 'C' packet and exiting the application leaves open the possibility that the packet will not be correctly received, but requiring an acknowledgement of the 'C' packet merely shifts the problem to that of the acknowledgement getting lost. It is envisaged that shutting the line down will be controlled by a higher level protocol indicating the end of an application and receiving an acknowledging message; both hosts, finding the line to be idle and knowing this means that both the end of application message and its acknowledging message have been safely received, can then close the link independently.

2.3.7. Packet interaction

2.3.7.1. Normal working

MMMS sends packets continuously, without waiting for a specific acknowledgement of each packet, and can have up to 7 sequenced packets outstanding (i.e. sent but not acknowledged) at any one time. In case of an error, the packet in error and all subsequent packets are retransmitted. Hence its classification as a Continuous-RQ protocol with Go-Back-N retransmission with a window size of 7.

Once the link has been initialised, there are three possible traffic states. These are idle, a message passing in one direction, and messages passing in two directions. In the first case the two hosts will merely periodically send an 'E' packet every time their T_2 timeout fires, and expect to receive such packets at

intervals sufficiently small to prevent T_3 from firing.

When sending a message, as in the cases two and three, message data is sent in sequentially numbered packets of either type 'M' or type 'L' (the latter if this is the last packet in the message). Packets are acknowledged using the acknowledgement number in the packet sequence character, which indicates the number of the next packet the receiver expects to receive; this implies that all previous packets have been received correctly. The acknowledgement can either be part of a message packet flowing in the opposite direction, as in the case where messages are passing in two directions at once, or an explicit 'E' packet if no message is passing in the opposite direction. In this latter case, the receiving host should send an 'E' packet in reply to nearly every packet correctly received; for example, the current implementation is constructed such that it periodically goes through a cycle of dealing with any incoming packets, constructing and sending any outgoing message packets, and finally sending an 'E' packet if it finds that it has received some packets and not acknowledged them. At low line speeds this produces an 'E' packet in reply to every packet received, but at higher baud rates where more than one complete packet may be received between each cycle then an 'E' packet may only be sent for every two or perhaps three packets correctly received.

Should the sender wish to abort the sending of a message, it sends an 'A' packet. This instructs the receiver to discard the message received so far and to return to a 'not receiving' state. 'A' packets are numbered and acknowledged in the same manner as 'M' and 'L' packets.

2.3.7.2. Problems and recovery

The basic problem to be handled is that of packets getting lost or corrupted; this boils down to the same thing, as any corrupted packet (i.e. one with an invalid checksum) is simply discarded. In this case the sequence of packets at the receiver will be broken when the next packet arrives. The receiver responds to an out of sequence packet by sending a 'N' (NAK) packet, and ignoring all 'M', 'L' or 'A' packets (i.e. sequenced packets) until one is received with a lowercase packet type 'm', 'l' or 'a' (these packet types are otherwise entirely equivalent to their uppercase counterparts). If no such packet is received after T_1 seconds then the 'N' packet is resent, up to a maximum of MAX_RETRIES times, after which the link attempts to reset (see below).

Further problems arise if acknowledgement packets are lost. Normally the next acknowledgement packet to be sent will be received soon afterwards, but this is not always necessarily so. For example, the sender may have sent a full window of packets and not received any of the acknowledgements, or the acknowledgement lost may be that to the last packet in the last outstanding message. To guard against this, if the earliest sequenced ('M', 'L' or 'A') packet has not been acknowledged after T_1 seconds then it and subsequent packets are resent, again up to a maximum of MAX_RETRIES times, after which the link attempts to reset.

To reset, the link discards any message being received and returns any message being sent (plus any messages sent but not yet acknowledged) to the top of the 'messages to send' queue. It then attempts to reinitialise as described in §2.3.5. Should this fail (perhaps by T_3 expiring — note that it is **not** reset for reinitialisation) then the link is closed down. If reinitialisation is successful then message transmission restarts from the top of the 'messages to send' queue, thus resending the message(s) in transit at the reset. Note from §2.3.5 the limit on the number of resets permitted.

It could happen that one or more messages are received correctly but the acknowledgement(s) for their last packet(s) are lost due to a reset. Hence the sender doesn't realise the message has been successfully sent and will retransmit them when the link is re-established after the reset. Up to seven packets may be outstanding at any one time, so in the worst case seven messages may be retransmitted. Therefore, if the sequence character of an incoming message matches that of one of the seven previous messages received, the new message is a duplicate and is discarded.

It may be wondered why MMMS insists on a specific reply to a NAK; after all, if the NAK gets lost the packet it is requesting will be retransmitted after a while anyway. The reason is that unless the reply to a NAK is identified, the NAK may be satisfied by a retransmission due to a timeout which occurred *before* the NAK arrived. The receiver then accepts any following packets until the retransmission generated by the NAK arrives unexpectedly. This generates a further NAK. Under some conditions this can lead a frequent exchange of NAKs, both sides being out of step but not realising it, and all attempts at readjustment failing. It is not possible for packets to be accepted out of sequence, but throughput is drastically reduced.

Finally, the receiver may receive the first packet in a new message but not have the buffer space to store the message. In this case it sends a 'S' packet, indicating that all packets after those acknowledged by the acknowledgement number in the 'S' packet will be ignored due to lack of space, and no further data packets should be sent until the receiver restarts the sender by sending a 'N' packet requesting the next packet. All subsequent data packets arriving provoke a response with the same 'S' packet, just in case the original one should have been lost.

Some examples of packet interaction are shown overleaf. Examples are given of a one-way transfer (Figure 2.13), a two-way transfer (Figure 2.14) and a two-way transfer over an imperfect line (Figure 2.15). The data within packets is not shown, just the packet type followed by the packet number and acknowledgement number. Initialisation and closedown sequences are not shown. Note that events on the same line in the examples are happening concurrently; for example, in the first example, the first 'E' packet sent by the receiver is sent as the third 'M' packet is being received, and hence the acknowledgement is only for the two packets already received. $\rightarrow$ and $\leftarrow$ highlight the direction of transfer of a packet, and $\rightarrow\bullet$ and $\bullet\leftarrow$ indicate a packet sent but lost or corrupted.

Sender	Receiver	Comments
M(0,0) →		Sender commences message
M(1,0) →		
M(2,0) →	← E(0,2)	
M(3,0) →		
M(4,0) →	← E(0,4)	
L(5,0) →		Last packet of message
	← E(0,6)	Final ACK

Figure 2.14: MMMS one-way transfer

Sender 1	Sender 2	Comments
M(0,0) →		Sender 1 commences
M(1,0) →		
M(2,0) →	← M(0,2)	Sender 2 commences
M(3,1) →	← M(1,3)	
M(4,2) →	← M(2,4)	
L(5,3) →	← M(3,5)	Last packet from sender 1
	← M(4,6)	ACKd by this
	← M(5,6)	
E(0,6) →	← M(6,6)	
	← M(7,6)	
E(0,0)	← L(0,6)	Last packet from sender 2
E(0,1)		

Figure 2.15: MMMS two-way transfer

Sender 1	Sender 2	Comments
M(0,0) →		Sender 1 commences
M(1,0) →•		Packet lost
M(2,0) →	← E(0,1)	
M(3,0) →	← N(0,1)	NAK packet 1
m(1,0) →		
M(2,0) →	← M(0,2)	Sender 2 starts
M(3,1) →	← M(1,3)	
M(4,1) →	•← M(2,4)	Packet lost
M(5,1) →	← M(3,5)	
N(0,2) →•	← M(4,6)	NAK sent but lost
M(6,2) →	← M(5,6)	○
M(7,2) →	← M(6,7)	○
M(0,2) →	← M(7,0)	○
M(1,2) →	← M(0,1)	○
M(2,2) →		Sender 2 window full
M(3,2) →	← M(2,3)	○ Timeout 1 — resend from packet 2
N(0,2) →	← M(3,4)	○ Timeout 2 — resend NAK
L(4,2) →	← m(2,4)	Last of Sender 1 message
	← M(3,5)	
E(0,4) →	← M(4,5)	
	← M(5,5)	
E(0,6) →	← M(6,5)	
	← M(6,5)	
E(0,7) →	← M(7,5)	
	← M(0,5)	
E(0,1) →	← L(1,5)	Last of Sender 2 message
E(0,2) →		

Figure 2.16: MMMS two-way transfer with imperfect line

Packets marked ○ are received by Sender 1 and their acknowledgement number noted, but they are otherwise ignored.

2.4. MMMS — Implementation and performance

MMMS is currently implemented as a library of C† functions running under MS-DOS on IBM PC or compatible microcomputers‡, VAX VMS and Berkeley 4.2 and 4.3 Unix. User programs should include the MMMS include file *mmms.h* and call the interface functions as necessary.

2.4.1. The MMMS library interface

There are 10 interface functions available. A summary of them follows, while detailed descriptions of each may be found in Appendix D in the form of Unix-style manual pages.

Function	Description
MS_Init	Set up the interface
MS_Start	Try to establish communication
MS_CloseDown	Abandon communication and dismantle interface
MS_SendMessage	Send a message
MS_ReceiveMessage	Receive a message, if any to receive
MS_Abort	Cancel any message being sent and clear the send queue
MS_GetConfig	Get link configuration info
MS_SetConfig	Set link configuration info
MS_LastError	Return error code of last error
MS_Status	Return state of the link

Table 2.8: MMMS interface functions

2.4.2. The library internals

The library is intended to be portable, and is accordingly divided into machine dependent and machine independent sections. The machine dependent portion of the code consists of eight functions

† C as defined in [Kernighan78] plus a few common extensions is used. Those extensions are structure assignment and the *void* type.

‡ Requires the SERINT serial port handler described in Appendix A.

collected in one module.

The code is intended to be portable between contemporary microcomputers, and is therefore constrained by the current rudimentary nature of most microcomputer operating systems. Ideally the protocol would be run as a separate server process, with the interface handling communication between the user process and the server. But only a small minority of current microcomputer operating systems are multitasking (though this situation is slowly changing), so something somewhat more unpleasant was called for. The implementation does require buffered interrupt-driven serial I/O, but this is not uncommon, and where it is lacking fairly simple to implement.

The typical microcomputer does have a source of periodic interrupts, and so the library code is implemented as a monolithic interrupt routine executed at least once a second. It has proved possible to move the code onto 4.3BSD Unix and VAX VMS, but it is doubtful whether other common operating systems will be as accommodating†.

The internal workings of the library are best explained in terms of the flow of message buffers illustrated below. These are held in a central pool and claimed either to store incoming messages or to buffer outgoing messages — Figure 2.16 overleaf illustrates.

When the first packet of a new incoming message arrives, an attempt is made to claim a message buffer from the pool. If none are left, a 'S' packet is returned telling the sender to wait until one can be obtained. Otherwise the message is decoded from incoming packets and placed in the buffer. When the final packet in the message has arrived and been incorporated, the message is complete. Its sequence character is then checked. If it is the same as one of the previous seven messages received, it is discarded as a duplicate. If it is the message expected, it is stored in a queue of messages received; when the message has been read the buffer is returned to the pool. Any other sequence character causes a fatal error — the link is closed.

When a message is presented for sending two buffers are requested from the buffer pool. If successful, the message is stored in one and the other returned. This ensures that messages to be sent cannot by themselves exhaust the buffer supply, but one buffer is kept free to receive messages (assuming that all

† Version 7 of Unix, for example, doesn't allow for timer signals with less than 1 second granularity.

buffers are not used storing messages that have been received). This prevents messages for sending from hogging all available buffers preventing reception of any messages.

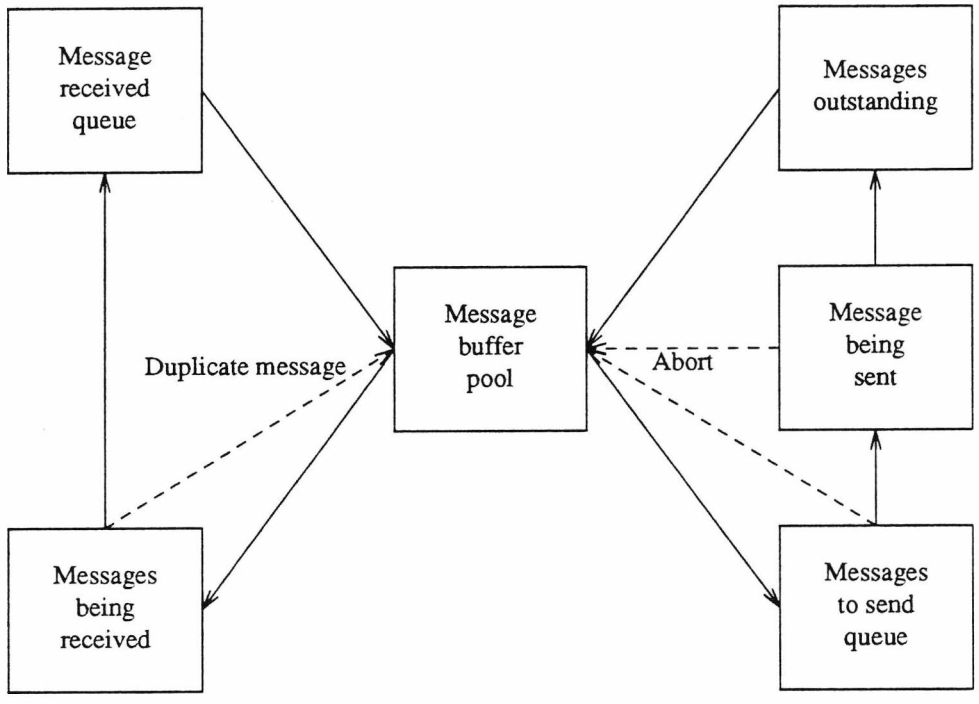

Figure 2.17: MMMS library buffer flow

The message is then placed in a queue of messages awaiting sending. When no message is being sent, the message at the top of the queue is removed and made the current message being sent. The message contents are then encoded into packets as space in the packet buffer becomes available. The packet buffer contains eight packets arranged in a circular fashion. Packets are alternately filled, sent and freed when acknowledged. Up to seven packets may be outstanding (i.e. sent but unacknowledged) at once.

Once the last of the message has been transferred to the packet buffer, the message is placed on a queue of messages dispatched but still in the packet queue. Each acknowledgement for a packet that was the last packet of a message causes the message at the top of this queue to be discarded and the buffer returned to the message buffer pool. If the link resets for any reason, the queue of messages awaiting acknowledgement is emptied and the messages placed at the top of the queue of messages to be sent.

2.4.3. Performance

Given a line requiring no padding or end of line characters to be sent with the packet, and data that requires no encoding, the overhead imposed by MMMS is 5 extra protocol characters required for every 94 data octets transmitted. So the theoretical efficiency of the protocol in this situation (i.e. the ratio of time taken to send the data without a protocol to the time taken with a protocol) is

$$94/(94+5) \approx 94\%$$

A simple file transfer program has been written that uses MMMS and the actual throughput under the above conditions was measured between two directly-connected microcomputers and a microcomputer and a VAX running 4.3BSD Unix.

2.4.3.1. PC — PC transfer

Two identical microcomputers (PCs)† were connected using a suitable cable and the time taken to transfer a file between them was measured. The results are given overleaf (Table 2.8) for various line speeds; for comparison, timings are given for the same transfer using Kermit. MMMS times are given for a one-way transfer (→) and a two-way transfer (←→); in the latter case the same file was sent in both directions simultaneously.

The file in question was composed solely of non-repeating upper case letters requiring no encoding, and was 13000 characters long. It was held in a RAM disc to minimise the effect of delays accessing backing store. All times are in seconds, and line speed is given in characters per second.

For lower line speeds the observed efficiency of MMMS is much as predicted above. As line speed increases, however, the efficiency falls off; in particular, doubling the line speed from 480 to 960 cps (characters per second) only gives a 67% increase in throughput rather than the expected 100% increase.

This calls for some explanation. The problem lies with the rate at which data is being transferred. MMMS handles serial I/O on the PC using the resident serial port handler described in Appendix A. Characters to be transmitted are placed in the handler's transmit buffer and dispatched to the port one at a time on receipt of an interrupt indicating the port is free. Similarly, characters are read from the port on receipt of

† Comcen IBM PC compatibles running MS-DOS 3.2.

an interrupt indicating a fresh octet is available. Considering just incoming characters, at a line speed of 960 cps a character arrives approximately once every millisecond.

File Transfer PC to PC						
Line	MMMS →		MMMS ←→		Kermit	
Speed	Time	Efficiency	Time	Efficiency	Time	Efficiency
960	18	75%	26	52%	36	38%
480	30	90%	31	87%	40	68%
240	58	93%	58	93%	71	76%
120	116	93%	116	93%	134	80%
30	460	94%	460	94%	504	86%

Table 2.9: MMMS PC to PC timings

The processor in a standard IBM PC, an Intel 8088, takes on average around nine clock cycles per instruction, and is driven at 4.77 MHz, giving an instruction throughput in the order of 0.5 MIPS (Million Instructions Per Second). So in the interval between characters arriving the processor will execute around 500 instructions. Somewhere in the order of 60 of these are used just responding to the interrupt and placing the new character in the receive buffer. To maintain full throughput, packets must be read from the buffer, checked for errors and decoded, reply packets must be built and added to the transmit queue, incoming data must be written to backing store and various system housekeeping functions must be carried out. To do all this in 440 instructions or less is almost certainly out of the question, so delays will occur waiting for packets to be read and replies generated, and throughput suffers. Following this analysis, two-way transfers should be subject to more severe degradation than one-way transfers; with input and output proceeding at full speed, the number of instructions executed between interrupts would be almost halved†. The observations bear this out. Doubling line speed from 480 to 960 cps increases throughput by only 19% for two-way traffic.

† Acknowledgement packets have to be transmitted during one-way transfer, of course. However these are only 5 octets long, as opposed to 101 in a full data packet, and impose a much lower overhead.

To confirm the above explanation, the measurements were repeated using different PCs, identical in every respect to the first pair with the sole exception that the clock speed was 8 MHz rather than 4.77 MHz, increasing instruction throughput by 68%. The efficiency of one-way transfer remained at 90% at 960 cps.

2.4.3.2. PC — Unix transfer

The procedure was repeated, this time between a PC and a minicomputer system running Unix‡. Communication took place over a terminal line linked to a UKCNET PAD; line speeds given are those from the PC to the PAD. At the time of measurement, the Unix system was servicing between 6 and 7 other users.

As before, times are given for one-way and two-way transfer. Two times for one-way transfer are given; transfer from the PC to the Unix system, and transfer from Unix to the PC.

Maximum packet length during transfers to and from Unix was limited to 40 data characters per packet; this is necessary to avoid swamping some UKC PADs. The reduction in packet length means that maximum theoretical efficiency goes down to 89%.

MMMS File Transfer PC to Unix						
Line	PC→Unix		Unix→PC		←→	
Speed	Time	Eff.	Time	Eff.	Time	Eff.
960	29	46%	29	46%	36	37%
480	34	79%	34	79%	47	57%
240	68	79%	65	83%	87	62%
120	122	88%	138	78%	161	67%

Table 2.10: MMMS PC to Unix timings

These measurements show MMMS performing considerably less well than over the direct PC to PC connection reported above. The loss of efficiency may be attributed to two factors; reduction in raw

‡ VAX 11/750/BSD 4.3

throughput due to traversing UKCNET and insufficient window size and/or packet size. This last is discussed further in §2.5.3 below.

Attempts have been made to quantify the effect that passage through UKCNET has on maximum possible throughput. Evidence amassed to date indicates that under the type of demands made on UKCNET by MMMS, raw throughput may in some circumstance be around 85% of that expected. In particular, on a 480 cps link, observed throughtput was only 414 cps. This would mean that expected protocol efficiency calculated as before would be in the region of 78% for line speeds of 480 cps. However, it must be stressed that these measurements are as yet inconclusive. Further work is required.

2.5. MMMS — Comments and Improvements

This section concludes discussion of MMMS. Various problems and/or improvements with MMMS are highlighted, and solutions examined.

2.5.1. Reliability

MMMS is intended to provide a reliable transport service, delivering data uncorrupted and in sequence. It is therefore vital that any packet corruption be detected; unfortunately, the current checksum mechanism cannot be considered at all adequate. Future versions of MMMS should adopt a 16-bit CRC as a minimum, preferably a 32-bit CRC. Use of a 16-bit CRC would reduce the probability of an error in a packet sent over a public dial-up telephone line remaining undetected to the order of 10^{-8} [Martin76]. Better yet, use of a 32-bit CRC would reduce that probability still further to the order of 10^{-13}; this would mean that on a 960 cps link transmitting packets continually, an undetected error would occur about once every 31,000 years.

Alternatively, or perhaps in addition, a CRC check could be generated over each message. This would additionally require some means of acknowledging receipt of messages and requesting retransmission of messages found to be corrupted. Given suitably reliable delivery of packets, this seems an unnecessary complication.

2.5.2. Portability

Portability here refers not to specific MMMS implementations, but rather to design aspects that make implementation impossible on some systems.

One important limitation is that MMMS as it stands is that it is restricted to working over full-duplex lines. This bars it from use with mainframes that operate only half-duplex terminal lines. This class of mainframes includes IBM and plug-compatible mainframes and so cannot be lightly ignored. One possible solution to this problem would be for the sender to relinquish control of the line when a complete window of packets has been sent, allowing the receiver to respond with ACK or NAK packets as appropriate. This would be using MMMS as an Idle-RQ protocol with a large packet size (7 normal MMMS packets would be sent, the sender would wait for an acknowledgement and so on).

The other important problem is the requirement for a link either supporting flow control or with a large buffer. A possible solution for lines without flow control would be to adopt the Zmodem approach of exchanging maximum buffer sizes during initialisation, limiting the number of packets outstanding at any time to a limit below that imposed by the protocol.

Lastly, although MMMS uses only a restricted character set, there is a case for restricting it still further. Some ASCII characters, including printable characters, do not map cleanly into EBCDIC, with the result that it is not uncommon for two translating systems to differ in the mapping used. There also remains the problem of needing one control character to pass unscathed. Further restricting the character set used to upper case and lower case letters, digits and a few selected punctuation characters would enable another printable character to be used as the start of packet marker.

2.5.3. Performance

To extract maximum performance from any Continuous-RQ protocol, it is important that data flows continuously, with minimum delays. Currently MMMS permits seven packets to be outstanding without acknowledgement; to ensure continuous flow, it is therefore vital that the acknowledgement for a packet reaches the sender before it has finished transmitting the sixth packet following. At a link speed of 960 cps, this leaves 0.6 seconds in which to generate an acknowledgement (this shrinks to 0.5 seconds if the ack-

nowledgement is piggybacked on a full data packet coming in the other direction, requiring 0.1 seconds to be transmitted). At 240 cps, the corresponding times are 2.5 seconds and 2 seconds respectively. These times can be easily exceeded by delays in a network and task-switching and I/O delays on the mainframe.

To improve performance in these circumstances, the number of characters that may be outstanding at any one time must be increased. There are two possible approaches.

1. *Increase packet size.* This offers the additional possibility of decreasing the packet overhead — the ratio of packet control characters to data characters — and hence improving protocol efficiency. However, it increases the time taken to recover from transmission errors, as retransmitting a packet takes longer. How much of a problem this is depends on the line speed and the error rate — if line speed is low and the error rate is high, longer packets may well degrade performance.

2. *Increase window size.* This would necessitate a larger ranger of packet numbers, and thus also require a revised packet format. It would also require increased packet buffer space, as would the previous option.

Other performance improvement measures involve reducing the amount of data that has to be transmitted. One possibility would be to replace the current system of quoting octets that cannot be sent as is. When sending random data, the quoting system imposes an average overhead of 0.7 characters for every octet transmitted. If instead octets were encoded by sending every two octets as three printable characters, the per-octet overhead would be reduced to 0.5 characters. Measurements taken on the set of VAX binary files used in §2.3.3 indicated possible savings in data transmitted of around 20%. However, the advantage of the quoting system is the low overhead it imposes on text, and measurements on the body of English text used in §2.3.3 indicated an advantage for quoting over 3-for-2 encoding in the order of 40%.

Reducing the amount of data to be transmitted may also be accomplished by compressing it. MMMS currently uses run-length encoding, which has the advantage of being trivial to implement and using no additional storage. The degree of compression produced is, however, inconsistent and in the worst case can expand the data to be sent by a factor of 1.5. A more powerful data compression method could offer the possibility of more consistent data compression. One such data compression method is adaptive Lempel-Ziv encoding [Welch84], which is reported to compress English text to 56% of original size, and binaries

by even more. If such compression could be achieved, encoding the resulting compressed data using the 3-for-2 method would still keep the amount of data to be sent through the link below the original size of the text. However, LZW compression performs best on long strings of data (circa. 6k bytes), requires storage space (30k+) for compression tables, and is not immune to expanding rather than compressing data. In addition, there is the question of whether a microcomputer could perform the required compression and decompression fast enought to keep the link busy. Measurements of microcomputer-based compression programs that use the LZW algorithm indicate that compression throughputs in the order of 2k bytes of compressed data per second are sustainable; however, this does not take into account the overhead of packaging and sending the data.

2.5.4. Facilities

MMMS currently lacks two important capabilities, namely multiple virtual circuits and expedited or priority messages. A simple form of the latter could be added with only minor changes to the protocol by maintaining two queues of messages for sending, a priority queue and a normal queue. If a priority message is submitted, any normal message being transmitted would be abandoned and returned to the top of the normal message queue. The priority message would then be sent. The only necessary change to the protocol would be a means of indicating to the receiver that this was a priority message.

Multiple virtual circuits would be a more complex addition, and one not felt to be in keeping with the aims of MMMS i.e. a simple means of providing reliable communication between applications.

This section has outlined several areas of concern and possible improvement in MMMS. The most important of these is undoubtedly the reliability issue — there is little point in upgrading the speed of delivery of corrupt data.

Chapter Three

The File Access System

This chapter begins with a discussion of the relative merits of bytestream files and record files for the problem under consideration, including a brief overview of the facilities provided by several contemporary record-based file systems. It then describes a record-based file access system, FIAP, intended for use over a reliable but slow (by file access standards) communication link, highlighting file formats and access methods supported and the available file access operations. The chapter concludes with a discussion of the deficiencies in the system as it stands and possible enchancements.

3.1. Records and Bytestreams

As noted above (§1.5), initial work on a file access protocol took the bytestream as the file model; that is, a file was viewed as merely a sequence of zero or more bytes. A variable number of bytes may be read from or written to the file in one operation, with a current file position being maintained, from which reading or writing is started. The bytestream model has the advantage of simplicity and generality — it is possible to impose any desired file and record structure on top of a bytestream, while with a record-based model a choice must be made from one of the (necessarily limited) file and record structure combinations available. Indeed, in a retrospective on the Unix operating system, Ritchie observes [Ritchie78]

> The notion of a ''record'' seems to be an obsolete remnant of the days of the 80-column card. A file should consist of a sequence of bytes. .... The greatest care should be taken to ensure there is only one format for files. This is essential for making programs work smoothly together.

This section explores the reasons why, despite these objections, the bytestream was rejected in favour of record-oriented interaction, and describes the record organisations available in three contemporary main-frame operating systems with record-based files.

3.1.1. Why records?

If the time spent performing file operations over a slow link is to be minimised, then the data sent over the link must be confined to that which is actually needed; any surplus is merely wasting link bandwidth. It must not be forgotten that the request for a file operation to be performed must also be sent

over the link. If the data being manipulated is in the form of fixed-length records then the amount and location in the file of any required data is known before the file operation starts. Consider, however, something as trivial as reading the second line of text held in a bytestream file, where the end of a line of text is indicated by a special 'end of line' character. If the file pointer is currently at the start of the file, we must read and discard characters until we find the end of the first line†.

Two possible approaches can be adopted; we can either read characters one at a time, or read the file in blocks of some suitable size and scan those once received.

The former option adds the overhead of transmitting a request and unpackaging the result for every read, and so, but for exceptional circumstances, the latter option is to be preferred. Consider, however, the case when the first line is blank and the second two characters long — if we read the data in blocks of, say, 512 bytes then we have needlessly fetched 508 characters.

Clearly, the ideal solution would be to be able to move to the start of the second line and send only those characters as are necessary. If a text file is regarded as a sequential collection of records with one line of text being a record, it is possible to request just the data required from the file server. So when dealing with any form of variable-length record, the amount of data that has to be transmitted can be minimised if the file server is capable of dealing in records.

As noted above, any desired record structure can be imposed on top of a bytestream. So, for example, an indexed file of variable-length records can be created and manipulated over a bytestream using appropriate software. However such manipulation is complex, and requires many separate operations on the bytestream. The cost in time when all these operations have to be performed over a slow link is prohibitive.

The above reasons for preferring record-based file operations over the bytestream model *for the case of individual operations taking significant time* centre upon making best use of link bandwidth. These are not the only reasons; there are also portability gains to be made.

There are a wide range of different formats in which mainframe record file systems store records, particularly variable-length records [Cunningham85]. For example, consider the representation of text — even

† We must also, of course, be alive to the possibility of encountering the end of the file.

on systems that employ bytestreams, conventions on how to indicate the end of a line differ. Record-based systems also have different ideas about storing text; as variable-length records of one type or another, or even fixed-length records padded with spaces or truncated as necessary. If we are to deal with mainframe files in the same manner as the mainframe operating system, we have two choices.

1. The microcomputer application is responsible for maintaining the correct format. This may include details like siting the start of records on disc block boundaries and other unwholesome particulars.

2. Restrict file operations to records and let the mainframe file server use the appropriate local format.

This latter option has the advantage that the same microcomputer application may be used with file servers on different mainframe systems.

3.1.2. Typical record facilities

This section examines the facilities provided by three record-based mainframe operating systems, namely DEC VAX/VMS, IBM MVS/370 and NCR ITX. In fact all three offer similar facilities, so after a description of VAX/VMS facilities the sections on MVS and ITX just highlight the differences between them and VAX/VMS. The section concludes with a look at record-handling facilities available for two bytestream-based operating systems, Unix and MS-DOS.

3.1.2.1. VAX/VMS

File and record access under the VAX/VMS operating system is provided by the VMS Record Management Services (RMS) subsystem [DEC86]. RMS services include both file processing (create/open etc.) and record processing (get/update etc.). The following is *not* a comprehensive summary of all RMS capabilities, but just an overview of the basic record facilities.

Records may be organised in a RMS file in one of three ways, although not all record types may be used with a particular file organisation. The three file organisations are

1. Sequential. Record follows record in the file with no gaps in between. This means that, for variable-length record files, the only way to determine the position in the file of any given record is to read every preceding record.

2. Relative. The position of each record in the file can be determined in advance. This means that variable-length records have to be stored within cells of fixed length.

3 Indexed. The contents of a fixed portion of each record are used as a key to that record, and an index is maintained which holds the record position and key for each record in the file. Secondary keys held in separate indexes may be specified, in which case file operations must indicate which index they are using.

Five different record types are available under RMS. These are as follows:

1. Fixed-length records. Each record in the file is the same length, that length being specified when the file is created.

2. Variable-length records. Rather than being fixed over the whole file, record lengths vary and are held in the file preceding each record. Two types of variable-length records are provided, format V (the default) and format D. Format V records store the length of each record in binary in two bytes, while format D records store the record length in decimal in four characters. The latter format is available only for reading and writing ANSI standard sequential magnetic tape files.

3 Variable-length records with fixed-length control (VFC). These are variable-length records, but additional control data may be stored in a fixed-length area of the record, the length of which is determined when the file is created. One suggested use for this control area is holding line-sequence numbers for each record in the file; these numbers could be used to locate records during file editing. VFC records are only available for sequential and relative files.

4. Stream records. Again, these are basically variable-length records but of different format to the above. No count of the record length is used; instead each record is terminated by a special character or character sequence. There are three variations — records terminated by ASCII CR, records terminated by ASCII LF, and records terminated by any character specified from a set including ASCII CR, ASCII FF, ASCII ESC and a sequence of ASCII CR LF. Stream records are only available for sequential files held on disc.

5. Undefined. Only available for sequential disc files, this format allows direct access to disc blocks.

Applications may use this format to implement their own record formats.

Finally, records may be accessed via one of four methods; a table of which methods are applicable to each file organisation is given below.

1. Sequential. Records are accessed in order from the beginning of the file for sequential or relative file organisations, or in index order in the case of indexed files. If an empty or deleted record is encountered in a sequential or relative file it is skipped.

2. Randomly by relative record number. Records in relative files are numbered from the beginning of the file (the first record being record 0). The relative record number can be used to specify the record to be processed.

3. Randomly by key value. Records in indexed files may be accessed by providing the key value of the record in question. The index is used to locate the record with that key.

4. Randomly by Record File Address (RFA). Whenever a record is accessed, RMS returns the record's RFA, the address at which that record is to be found on disc. That address can be used to return to the record; this is the only method of accessing records in a variable-length record sequential file randomly.

	File organisation		
Access mode	Sequential	Relative	Indexed
Sequential	Yes	Yes	Yes
Random by relative record no.	Yes†	Yes	No
Random by key value	No	No	Yes
Random by RFA	Yes	Yes	Yes

Table 3.1: RMS file access modes

Note that random access is only possible in files held on disc. Sequential access to a file is always possible;

† Fixed-length records on disc drives only.

it is also possible to mix random and sequential accesses. RMS maintains a *file pointer* which indicates the record that will be affected by the next sequential operation. Accessing a record randomly sets that file pointer to the record following the record accessed.

It should also be noted that, depending upon file organisation, certain file operations may be restricted. For example, the addition of records to a sequential file is only permitted if the file pointer is positioned at the end of the file.

3.1.2.2. IBM MVS/370

The record options provided by the MVS Data Management Services [IBM83] are broadly similar to those supported by VMS.

Fixed-length records (format F) and variable-length records (formats V and D) are as described in the previous section. In the case of format D, MVS translates ASCII to or from the EBDCIC character coding used internally in IBM mainframes. MVS does not support stream records or VFC records, though the first character of each record may be an optional control character. MVS does have 'undefined' records (format U), which correspond to VMS undefined records, treating each file system block as a record.

MVS file organisations include sequential, direct (akin to relative) and indexed. There is also a fourth organisation, partitioned files. These consist of independent groups of sequentially-organised records, known as members, indexed by member name. This file organisation is generally used to hold code libraries in either source or compiled form.

As mentioned, direct files are akin to VMS relative files. But whereas records in VMS relative files may be retrieved by relative record number, MVS direct files can only be accessed by the record address (cf. VMS RFAs). Furthermore, the facilities provided for accessing direct files only exist at the file system block level, making them a good deal more painful to use than indexed or sequential files; record-based access facilities are provided for the latter two as well as file system block access functions.

Two forms of indexed files are available under MVS; both may be accessed by key or sequentially. The older format, Indexed Sequential Access Method (ISAM) files may only have one index and must periodically be restructured as records are added to maintain access speeds. Virtual Storage Access Method

(VSAM) files are also available — these do not require this periodic restructuring and permit the use of more than one index.

3.1.2.3. NCR ITX

The record options supported by ITX [NCR86] form a subset of those provided by VMS.

Fixed-length records (format F) and variable-length records (formats V and D) are all available. ITX does not support stream records, VFC records or undefined records.

ITX file organisations are sequential, relative and indexed. As with MVS there are two formats of indexed file available, with differing capabilities.

Records in ITX relative files are accessed by relative record number only. ITX does not support an equivalent of the VMS RFA.

The two ITX indexed file formats are known as *indexed* format and *alternate key* format, the latter being the newer.

Indexed files offer only a single key index, stored in the same file as the record data. The maximum number of items in the file must be specified when the file is created, and index space grows down from the end of the file.

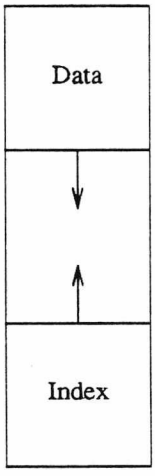

Figure 3.1: ITX indexed file layout

The file will therefore need restructuring should the volume of data exceed expectations. ITX does not

reuse space released by deleted records, so if record deletion is commonplace the file will need periodic restructuring.

The newer *alternate key* format allows up to 15 secondary keys. All key indexes are held in separate files, so file growth is not the problem it is with *indexed* files. Space taken by deleted records can be reused, and files do not require periodic restructuring.

3.1.2.4. Records on bytestreams

The X/Open group, a group of predominantly European computer manufacturers, has been engaged since 1984 in defining a Common Applications Environment. This environment, based on AT&T Unix System V, is intended to allow conforming applications to be portable at the source code level across systems produced by participating manufacturers, thus increasing the application base available to each manufacturer. The introduction to the X/Open Portability Guide [X/Open85] observes

> The input/output facilities supported by System V consist only of bytestream read and write operations on files. No facilities are provided for operating on files as sets of records. This leads to application writers having to make their own arrangements for record handling, resulting in both a multiplication of effort and a proliferation of non-standard methods.

Accordingly, the Common Applications Environment specifies an interface to an ISAM file handling library.† This supports fixed-length record indexed files, allowing random access by key and sequential access in index order. Secondary keys are supported; indexes for these and the primary index may or may not be held in separate files, depending upon the implementation. Records may also be accessed sequentially in physical order, or randomly by relative record number. Variable-length records are not supported, and it is not possible to manipulate indexes separately e.g. to implement variable-length record ISAM files.

The programming effort involved in handling sequential fixed-length or variable-length records over a bytestream is not great. The effort involved in implementing indexed files, however, is. Accordingly several suppliers‡ now offer libraries providing either full ISAM facilities or B-Tree implementations (the B-Tree [Comer79] is the basis of modern indexing systems that permit random and sequential access).

† The interface is in fact based upon the interface presented by Relational Database Systems Inc.'s C-ISAM library.

‡ At the time of writing, one software distributor is offering no less than 15 such products for MS-DOS.

Two examples of MS-DOS libraries are the Zortech Database Package [Zortech87], for C programming, and the Borland Database Toolbox [Borland85] for use with Borland's Turbo Pascal. Both include full source code; their facilities are compared below.

The Zortech Database Package offers four distinct libraries. These provide fixed-length record handling, B-Tree indexes†, variable-length record handling and an interface to the first two that provides convenient manipulation of fixed-length record ISAM files — indexes are updated automatically, for example. All indexes must be held in separate files.

The Turbo Database Toolbox is similar in concept, but provides only fixed-length record handling and B-Tree manipulation. No unifying ISAM library is provided — the programmer is responsible for updating all indexes.

The record-handling facilities provided by both the above libraries allow records to be accessed by relative record number. In fact, the B-Tree handling routines that both provide store the record key and relative record number in the B-Tree; to retrieve a record, the B-Tree library is used to find the relative record number of the record with the given key.

3.2. FIAP overview

Referring back to the file access system overview given in §1.5, the next requirement is for the file access sections of the system, namely the interface library, the file server and the protocol linking the two (see Figure 3.2 overleaf).

The combination of interface, server and protocol described below is named **FIAP** (**F**ile **A**ccess **P**rotocol). The remainder of this section gives an overview of FIAP, describing its design aims, the available file organisations and record types, and the file manipulation operations provided.

† In fact the fixed-length record handling routines are supplied as part of the B-Tree manipulation library, as it makes use of the routines.

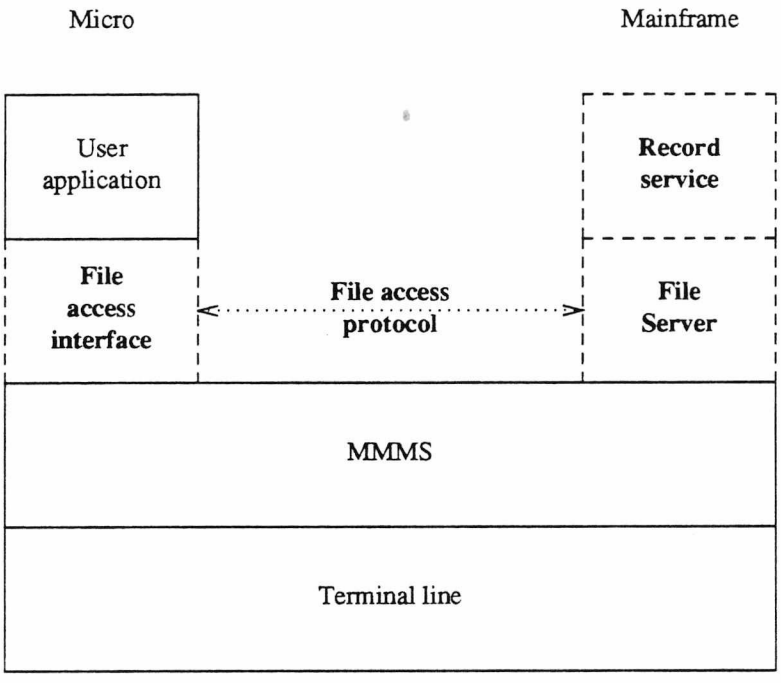

Figure 3.2: File access system overall design

3.2.1. Design Aims

This section summarises the various factors influencing the design of the FIAP system, considering the server, the application interface† on the microcomputer, and other miscellaneous areas including the protocol used between server and microcomputer. It concludes by highlighting areas not addressed by FIAP.

First and foremost, although FIAP is initially to be run using MMMS as the source of the transparent error-free protocol link it requires, it should not be tied to MMMS in any way, but should be free to use any other suitable transport mechanism.

3.2.1.1. The file server

For the sake of simplicity and portability the file server should be a normal user process, started before an application using FIAP is initiated on the microcomputer and terminating when the application

† That is, the interface the FIAP library presents to application programs making use of it.

exits (this is analogous to the way Kermit works). This has two benefits. Firstly, because the process is owned by the user, it is automatically subject to the normal restrictions on what files may be accessed by the user; and secondly this does not require the host operating system to be modified in any way.

The file server should make use of local record formats and file organisations as far as possible, so that files manipulated using FIAP remain generally accessible to other programs residing on the mainframe.

The server should support a standard set of file organisations and access methods — local record handling facilities may be used if possible, but they should not restrict the server capabilities. The server itself must supply missing functionality. This may in some cases contradict the previous requirement; for example, the local indexed file capabilities may not be up to providing the indexed file requirements of FIAP. In this case, the requirement for a standard set of file facilities takes precedence; one of the main motivations behind FIAP is, after all, that the mainframe should carry out as much file processing as possible.

Finally, some applications require temporary files with unique names sited in scratch filespace areas. The server should provide these, automatically deleting them when they are closed.

3.2.1.2. The application interface

The ideal interface to FIAP facilities would be one identical to that presented by the file handling section of the microcomputer operating system. This is not possible for two main reasons, the first (and most obvious) being that a microcomputer operating system is unlikely to provide the file facilities offered by FIAP. The second has to do with the time taken by FIAP operations. Whereas a local file operation will complete in a short time, FIAP operations can take a significant time to complete due to the low bandwidth of the link to the server. This time could be put to profitable use by the application. The application interface should not require the user program to surrender control for significant periods. Finally, a different interface emphasises to the application programmer that the time taken by file operations must be considered carefully.

The vast majority of microcomputer file systems are based on the bytestream model; this includes that of MS-DOS, by far the most prevalent. Although FIAP is based on record access, the application interface should be close in style to the MS-DOS file interface, as it is one with which applications programmers

are likely to be familiar.

Application programs should not have to concern themselves with the formats in which records are stored. This should be entirely taken care of by the file server, thus allowing microcomputer FIAP applications to be portable amongst mainframe operating systems equipped with a FIAP server.

Finally, to promote portability of the application interface between microcomputers, the interface should not assume any form of multitasking is available, as this is still uncommon in microcomputer operating systems. This means the application cannot rely on any (un)buffering and other bookkeeping operations taking place without it occasionally requesting them. This does not preclude such things happening automatically under a suitable operating system; rather it provides for cases where they cannot.

3.2.1.3. Issues not addressed by FIAP

FIAP does not attempt to address several file and/or portability concerns. This section merely highlights the omissions; §3.4 below comments further on how some of these might be addressed in future versions.

— Mainframes with byte lengths other than 8 bits are not catered for at all.

— There should be some means by which file servers can automatically translate characters to and from ASCII if the mainframe operating system uses another character encoding.

— No attempt is made at translation of filenames to conform to local mainframe conventions, though the provision for the automatic generation of temporary file names acknowledges the problem.

— No facilities are provided for modifying access permissions or manipulating directories in any way.

— No form of file or record locking is specified in FIAP. File servers are not precluded from adopting some form of locking; applications should have the facility to manipulate file or record locks.

3.2.1.4. FIAP and the OSI Reference model

FIAP can be fitted into the ISO OSI Reference Model as follows:

OSI	FIAP
Application layer	File access functions
Presentation layer	FIAP file organisations are mapped into local equivalents. No file content mapping is done.
Session layer	Requests are scheduled and results received. The link with the server is established and broken. Communication failures are noted, but FIAP makes no attempt to recover from them.

Figure 3.3: FIAP and the OSI Reference Model

3.2.2. Record types and file organisations

Drawing on the observations of §3.1, FIAP supports fixed-length records and two variable-length record formats, one of which is the 'standard' format and the other a portable format (c.f. format V and format D). The latter format will not necessarily be different to the standard format — it depends on the mainframe file system. Note also that FIAP does not specify the record formats explicitly; this again is a matter for the file server, as specified in §3.2.1.1.

§3.1.1 observes that even among record-based filing systems, the representations used for text differ. FIAP therefore also defines a 'text' record format, a variable-length record format where one line of text corresponds to one record. Again, the file server uses the most appropriate format; application programs using FIAP do not have to be concerned with end-of-line conventions, padding lines to a fixed length or whatever devilry is required by a particular mainframe file system.

FIAP specifies three different file organisations; sequential, relative and indexed. Not every record type may necessarily be used with a given organisation; details are given below.

Sequential files are supported for every record type. A sequential file of text records should be the format used by normal text files under the mainframe filing system in use. Alternative-format variable-length records are not available in any other file organisation.

Relative files are supported only for fixed-length and standard format variable-length records†.

Indexed files are available for fixed-length and standard variable-length records. They may be accessed either randomly by key or sequentially in key order. In addition to the primary key, a number of secondary keys may also be specified when the file is created. Keys are composed of one or more segments of characters taken from the record (there is a restriction on overall key length); should a segment require characters that are either past the end of the record or are not printable ASCII characters, the offending characters are replaced with spaces for key comparisons. Duplicate keys are only permitted if specified when the file was created, and if permitted the maximum allowed key length may be reduced. The order in which records with identical keys are retrieved is not defined.

FIAP also provides a file organisation designed to be useful in work files for applications programs that manipulate text. This file organisation, referred to as relative text in Appendices B and C, is more properly thought of as an indexed format, where variable-length records are indexed on the line number of that record in the text. When records are added or deleted line numbers change as appropriate. For example, consider the file below (the line numbers are included for illustration only).

1 There was a Baboon,
2 who one afternoon
3 with two great palms
4 strapped to his arms
5 waited for the wind to drop before
6 he started his take-off run.

Deleting record 5, rewriting record 3 and adding a new record before record 3 results in

† Readers who have peeked at Appendix B or Appendix C will have found reference to relative text files — these are more properly described as indexed files and are discussed below.

1 There was a Baboon,

2 who one afternoon,

3 said 'I think I will fly to the sun.'

4 So with two great palms

5 strapped to his arms

6 he started his take-off run.†

This relative text format is only intended for use in work files, as it is unlikely to exist as a standard file format.‡

The following table shows the available record format — file organisation combinations.††

	File organisation		
Record format	Sequential	Relative	Indexed
Fixed-length	Yes	Yes	Yes
Variable-length	Yes	Yes	Yes
Alternative variable-length	Yes	No	No
Text	Yes	No	Yes

Table 3.2: FlAP record format/file organisation combinations

3.2.3. File Access Operations

This section gives an overview of the interface FlAP presents to an application program. The only current FlAP implementation comprises a Unix-based file server and an interface library written in C for IBM PC series microcomputer running MS-DOS. Precise details of the interface may be found in Appen-

† From "Silly Old Baboon", [Milligan71]

‡ It should be added that in the current FLAP implementation, relative text files are not at all robust; one disc error is likely to render the whole file unreadable.

†† Readers browsing through Appendices B and C will also find mention of a sequential bytestream format. This is a format intended to allow access to the raw file system at the most basic level. It is currently experimental (and badly-named — the underlying file will not always be a bytestream).

dix C in the form of Unix style manual pages.

Firstly, a word on filenames. As noted above (§3.2.1.3), FIAP makes no attempt at converting filenames to conform to local conventions. There is one exception — file servers may truncate filenames to the local maximum filename length if appropriate. FIAP also imposes an outright limit on filename length of 255 characters.

The basic FIAP file model is intended to resemble the bytestream model used in MS-DOS and Unix, with the smallest unit of access being the record rather than the byte. FIAP files are manipulated via a *file descriptor*, a small integer obtained when the file is opened. Each open file has a currency, the current position in the file from/to which reading/writing will take place. The currency is affected by each read and write, and may also be read and set.

The currency in an MS-DOS/Unix bytestream is specified as the number of the current byte — the one that will be affected by the next read or write — in the file. The first byte in the file is byte number zero. Similarly, the first record in a FIAP file is record zero, and the current record is that record which will be read or written next, assuming currency is not altered. In some circumstances when using FIAP the current record number is unknowable, e.g. after a random access by key to a record in an indexed file. Currency still works as normal; only the record number is unobtainable.

All FIAP functions that have to interact with the server may take time to complete. The application interface, therefore, just *initiates* the sending of parameters and reception of results and returns control to the application at once. The application must then make periodic checks to see if the operation has completed. Currently only one FIAP operation can be outstanding at any one time. All operations return both a result and an error code. In the current implementation, the latter is made available via a global variable.

The functions provided by the FIAP application interface may be classified into five groups as follows.

1. *Management functions*. These look after establishing communication with the server and monitoring any ongoing operations.

2. *Basic file functions*. The basic functions used in manipulating files.

3. *Multiple record functions*. These perform one of the basic file functions but on several records at once. Requesting, say, four records be written at once is quicker than requesting four individual writes because it is not necessary to wait for each write to finish and results to be returned before the next can get underway.

4. *Extended operations*. These carry out 'higher level' file processing functions at the server.

5. *File management functions*. Depending on the implementation, one FlAP file may be actually implemented as several files. For example, in the current implementation, indexed files are implemented as a data file and as many index files as required. These functions allow basic file management to be done in a way that treats these multiple files as single objects.

The rest of this section gives a brief summary of each function. As a convenience for the programmer, all *flq?????* functions have an equivalent named *fl?????*, which queues the request and waits for the result to be returned.

3.2.3.1. Management functions

flqAbort Several FlAP operations deal with more than one record. flqAbort sends a request for the function to be abandoned after the next record has been processed. The result of the aborted function must be collected in the usual way.

flDone Monitor progress on ongoing FlAP operation. If operation is complete, return operation result and error code.

flInit Initialise the FlAP library. Must be called before any FlAP operations can be used.

flReady See if communication with server has been established.

flShutdown Close down the FlAP library. Must be called to close down the server and communication channel.

flStart Try to establish communication with server.

flWait Wait for ongoing FlAP operation to complete.

Any application using FIAP must establish contact with the server before trying any file operations. This is done by first calling *flqInit* to initialise the interface library, then *flqStart* to start communication attempts. *flqReady* is then used to monitor connection until either the file server responds or the connection times out (the application may of course unilaterally give up on the connection before the timeout is triggered). Note that *flShutdown* must be called before the application exits; the current implementation will crash if this is not done, as shutting down MMMS tidies up various asynchronous signalling routines.

3.2.3.2. Basic file functions

flqAddRec Add a new record. If the file is indexed, the key is inserted into all file indexes. Relative text files add the record *before* the current record. All other file types add the record at the end of the file.

flqClose Close a file descriptor. If the file was a temporary file, it will be deleted automatically.

flqCreate Create a new file. The record length and file type must be specified (also key specifications if the file is indexed).

flqDelRec Delete one or more records. This function can also be classed as a multiple record operation.

flqFindRec Locate a record in an indexed file using random access by key in a specified index, and read that record if found. This function can also be used to select an alternative index without searching for a record. In indexes which allow duplicate keys, the first occurrence of the key is found.

flqGetRec Read a record.

flqOpen Open an existing file for processing. The file type and record length must be specified.

flqSeek Alter file currency by moving to a new record, specified by the offset from the start of the file (i.e. the relative record number), from the current record or from the end of the file.

flqTruncate Truncate a file from the current record onwards.

flqUpdateRec Rewrite an existing record. Updates all key entries if in an indexed file. If the currency was at the end of the file, then this adds a new record.

Adding a record to a text relative file inserts the new record before the current record. The distinction between adding new records and updating old ones must be drawn specifically for use with these files, and also with indexed files which permit duplicate keys, as otherwise it is impossible to know whether, given a record, to add it as new with a duplicate key or to replace an existing record.

Records in all types of file may be accessed by relative record number (the relative record number of a record in an indexed file is its position in the currently-selected index). However in sequential or indexed files this can involve scanning through intervening records first, and so may not be a sensible course of action.

3.2.3.3. Multiple record operations

flqVAdd Add multiple records. Equivalent to multiple calls to flqAddRec, but quicker.

flqVGet Read multiple records. Equivalent to multiple calls to flqGetRec, but quicker.

flqVUpdate Rewrite multiple records. Equivalent to multiple calls to flqUpdateRec, but quicker.

These operations are equivalent to multiple calls to the corresponding single record operation, with each call being preceded by a call to *flqSeek*. Should any part of an operation on a record fail, the multiple record operation aborts, indicating the number of records successfully processed.

3.2.3.4. Extended operations

flqCompare Compare records in two files until they either match or differ.

flqCopy Copy multiple records from one file to another.

flqDuplicate Duplicate multiple records in a file. This is equivalent to reading all the records to be duplicated into temporary storage using flqGetRec, and then adding them using fqlAddRec.

flqSearch Search through records until one with contents matching the search pattern is found.

The search pattern supplied to flqSearch is a Unix-style regular expression as used in the Unix utilities *grep* and *ed* and described in [Kernighan78].

3.2.3.5. File management functions

flqDelFile Delete a file and any separate but related index files.

flqRenFile Rename a file and any separate but related index files.

3.3. FIAP implementation

This section describes several aspects of the current FIAP implementation, namely the protocol the interface library uses to pass requests to and receive results from the server, the record service used by the server and finally a protocol used by the mainframe to instruct a suitably-programmed terminal program on the microcomputer to load and execute applications programs. This latter mechanism allows applications using FIAP to be started from a mainframe command line.

3.3.1. The server protocol

As mentioned above (§3.2.1), FIAP presupposes a reliable, transparent link between the application interface and the file server. This is provided by MMMS in the current implementation. This section describes the simple protocol the server uses to receive requests and return results.

Every FIAP operation consists of an operation request, possibly followed by operation parameters, being passed to the server, which returns an operation result and an error code, optionally preceded by result data. Some potentially time-consuming operations may be aborted while in progress.

The data values exchanged are of three main types. The basic type is the octet, which is also the unit of communication with the transport link. The other types are sent as multiple octets; these other types are shorts and longs. More complex data structures are made up from combinations of these three types.

Shorts and longs are 16-bit and 32-bit quantities respectively. They are exchanged using 2 and 4

octets, sent least-significant octet first. This is easiest to explain by example. The long value 05F5E100 hexadecimal (decimal 100 million) is split into four octets 05 F5 E1 00. These are sent in the order 00 E1 F5 05. Similarly the short value 2710 hexadecimal (10,000 decimal) is split into two octets 27 10 and sent 10 27.

Two common examples of more complicated data structures are filenames and record data. Both are sent as multiple octets preceded by a count of the octets being sent (not including the count). Filenames being restricted in length to 255 characters, their count is sent in one octet (this being the motivation behind the restriction). Records, on the other hand, may exceed 255 octets in length, and so their count is sent as a short.

The protocol exchanges packets of octets over the transport service, which must allow packets of at least 25 octets to be transmitted. The first octet of any packet identifies the function of that packet. FIAP operations are each assigned a code from ASCII 'B' to 'T' inclusive. Other packet identifiers are '&', indicating a continuation of data in the previous packet (multi-octet values, including shorts and longs, may be split over packet boundaries), '=' indicating the first packet of result data returned by the server (if any), and 'A' requesting the operation be aborted.

The general form of a request to the server and the response is shown overleaf in Figure 3.4. '@' is used as the FIAP operation identifier — it does not correspond to any actual operation. Three notational conveniences are used; packets enclosed in [..]* occur zero or more times, while packets enclosed in [..]? occur at most once. Descriptions of packet contents are enclosed in {..}.

If the operation in progress is aborted, sending of further parameters is abandoned and an abort ('A') packet is sent. On receipt of an abort packet, the server ceases any ongoing activity and returns the result and error code packet immediately. Should an abort packet arrive after the result packet has been sent, the likely reason is that an operation was aborted before the first packet had been transferred. Since a result for the operation is expected, the server returns a result packet indicating the operation was aborted.

Application Interface		File Server
@{parameter data}	→	
[&{more parameter data}	→	]*
[A	→	]?
	←	=[{result data}]*
	←	&[{more result data}]*
	←	@{result code}{error code}

Figure 3.4: FIAP packet interchange

The protocol simply allows the application interface to make remote procedure calls on the file server. These calls are exactly as listed for the application interface; its sole job is handling protocol traffic. This means the server is *stateful*; that is, it retains information on which files are open, what their file descriptors are, and so on. In the event of a breakdown in the transport service, no attempt is made to re-establish communication; the server closes down and the application interface returns a 'communication lost' error.

Full details of the various packet types and parameter data are given in Appendix E. As an illustration of actual packet transactions, two examples of protocol traffic are given below. First, a simple example — closing a file. The file descriptor of the file being closed is a a byte quantity, here F. The result code returned is R and the error code is E. The operation code for close is D. Long values are subscripted L, short values S. Actual characters sent are enclosed in ''.

Application Interface		File Server
'C'F	→	
	←	'C'R_LE

Figure 3.5: FIAP packet traffic — closing a file

Now a more complicated transaction. The application requests N records to be read, but the operation is aborted after only one has been returned. The file to be read has file descriptor F; the record to be read has contents C and length L. The records are to be read in sequence. Multiple record operations specify an implied seek for each record to be processed, supplying the record offset and the offset type. The required values of offset and offset type for reading records in sequence are 0 and 1 respectively†. Finally, buffers of size B have been allocated to receive each record. The maximum amount of data returned is this number of octets, even if the record is actually longer. The operation code is M.

Application Interface		File Server
'M'Ns0L1Bs0L1Bs0L1Bs	→	
	←	'='LsCCCCCCCCCCCC
	←	'&'CCCCCCLsCCCCCC
'&'0L1Bs0L1Bs0L1Bs	→	
'A'	→	
	←	'M'RLE

Figure 3.6: FlAP packet traffic — aborting a multiple-record read

3.3.2. The URAT record library

§3.1.2.4 notes that the Unix file system is based on the bytestream model and offers no record handling facilities. The only current FlAP file server is, for divers reasons, Unix based and uses a specially-written library for record access. This library, URAT‡, provides a generally useful record facility; this section describes the operations provided and highlights the efforts made to fulfill the requirements described in §3.2.1.1.

URAT provides all the file types required by FlAP, as listed in §3.2.2. Files may also be flagged as

† See Appendix C for details.

‡ Unix Record Access RouTines. Pronounced 'you rat'.

temporary, in which case they are placed in the Unix scratch filespace under */tmp* and deleted on being closed. Indexes are held in separate files with filenames derived from the given filename by appending ".ix" and the number of the index, counting the primary index as index one. Variable-length record indexed files also have an extra index, ".ix0", holding details of where the record may be found in the main file.

The key specifications of indexed files are held at the start of the data file for convenience. By the same token, the record length (or maximum record length) of the other file formats should be held in a similar manner, removing the possibility of incorrectly specifying them when existing files are opened. However, this is not done. §3.2.1.1 stressed the need to use local record formats whenever possible. Considering a fixed-length sequential record file as a sequence of records in the file, it was considered advisable to use the most obvious format — for record of length *l*, record number *n* starts at byte offset *nl* in the file. Hence when opening an existing file the record length must be supplied.

Similar reasoning applies to variable-length files. These are implemented in two forms, format V and format D. Relative variable-length record files are available in format V only and place records in fixed-length cells á là VAX/VMS.

Records in sequential text files are delimited by newline characters, the standard Unix mechanism for indicating the end of a line. When reading text files, records are also terminated by the end of the file. It is not possible to write a text file that does not end in a newline.

The record access operations provided by URAT are, like those of FIAP, modelled on those used to manipulate Unix bytestreams, with the smallest unit of access being the record rather than the byte. Unsurprisingly they form a subset of the FIAP functions, with two related exceptions. A summary of the available operations is given below; full details may be found in Appendix B.

urAddRec Add a new record. If the file is indexed, the key is inserted into all file indexes. Relative text files add the record *before* the current record. All other file types add the record at the end of the file.

urClose Close a file descriptor. If the file was a temporary file, it will be deleted automatically.

urCreate	Create a new file. The record length and file type must be specified (also key specifications if the file is indexed).
urDeleteFile	Delete a file and any separate but related index files.
urDeleteRec	Delete the current record.
urFindRec	Locate a record in an indexed file using random access by key in a specified index, and read that record if found. This function can also be used to select an alternative index without searching for a record. In indexes which allow duplicate keys, the first occurrence of the key is found.
urGetCurrency	Save the file currency.
urGetRec	Read a record.
urOpen	Open an existing file for processing. The file type and record length must be specified.
urRenameFile	Rename a file and any separate but related index files.
urSeek	Alter file currency by moving to a new record, specified by the offset from the start of the file (i.e. the relative record number), from the current record or from the end of the file.
urSetCurrency	Set the file currency to a position previously saved by a call to urGetCurrency.
urTruncate	Truncate a file from the current record onwards.
urUpdateRec	Rewrite an existing record. Updates all key entries if in an indexed file. If the currency was at the end of the file, then this adds a new record.

Alert readers will have noted the inclusion of two functions not related to FlAP functions, namely urGetCurrency and urSetCurrency. They provide a means of saving and restoring file currency when the current record number is unknown, a frequent occurrence when using indexed files. Strong consideration was given to adding equivalents to FlAP — their utility is demonstrated by their frequent use in the FlAP file server — and they would probably appear in any future version. The problem is that each file server would almost certainly require a different 'magic cookie' to be able to restore currency fully, and a suitably

general format has yet to be defined.

3.3.3. Application starter protocol

To enable FIAP based operations to be started from a mainframe command line and hence be integrated into the mainframe environment, a simple protocol has been designed which can act as a prologue to a FIAP session. This assumes that communication takes place over a normal terminal connection; but then that is the environment at which FIAP is aimed.

The alternative approach would be to place control at the microcomputer end of the line. Although possible, this approach is more difficult to implement seamlessly because the complexities of logging on and so forth must still be dealt with. Frequently this can be automated, but users must still be aware of user names, passwords and such. Conceptual switches must be made between mainframe and microcomputer environments. Controlling from the mainframe environment offers the possibility of a seemingly consistent environment with a minimum of implementation effort, because the process of establishing communication and logging on is part of that environment. Implementation is easier because the mainframe merely has to request, through an established communication channel, the execution of the required microcomputer-based application program and pass the necessary parameters. When the program has finished, normal terminal emulation is resumed.

The protocol described below, in conjunction with a suitable terminal program on the microcomputer, does just this. Some attempt is made to ensure reasonable behaviour if the terminal does not support the protocol, or has no knowledge of the application program in question. Command line parameters are split into two types, flag and parameters, in a rudimentary attempt to alleviate problems with differing conventions among mainframe operating system command line interpreters on how flags/switches are handled.

The operation of the protocol is as follows. The mainframe starts the request for a FIAP application to be run by sending the following sequence. ESC, BEL and CR are the ASCII control codes. Items between '' are sent as is.

ESC BEL 'FIAP: FIAP operations not supported. Press RETURN.' CR

If the microcomputer recognises the first seven characters, it should reply immediately with

'FIAP:OK' CR

If any other sequence is received or a timeout T expires the operation is abandoned. On successful receipt of the reply, the name of the requested program, flags and parameters are then sent as below

'FIAP:' program-name BEL flag-1 BEL flag-2 BEL ... flag-n

BEL BEL param-1 BEL param-2 BEL ... param-n CR

If the microcomputer does not receive such a sequence before a timeout T, again the operation should be abandoned. If all goes well, the microcomputer should check that the requested program is available, and if so reply with the same sequence as before. If not, the operation should be abandoned (e.g. send just a CR).

The value of T must be long enough to prevent a timeout caused by the time taken for the various messages to negotiate a slow terminal connection. The value used currently is 10 seconds.

Once a successful reply is received from the microcomputer, the mainframe then starts execution of the FIAP file server; the microcomputer meanwhile is loading the application and the two will then establish contact and proceed.

The control character sequences used have been chosen because it is unlikely they will be filtered out by the mainframe operating system and also unlikely they will occur in normal terminal traffic.

The above protocol is intended for use in conventional asynchronous ASCII terminal environments. Should FIAP be used in more esoteric environments some alternative may be preferable. Of course, the user can always start the server and the application program manually, so as to speak, by starting the former from the mainframe command line, exiting the microcomputer terminal program and starting the application from the microcomputer command line.

3.4. Reflections

This section concludes the chapter on FIAP with some comments on various aspects of FIAP, discussing perceived weaknesses and possible improvements. These are roughly divided into three categories, file operations supported, the protocol and finally the file organisations supported.

3.4.1. File operations

FIAP currently provides two file management, as opposed to file access, operations, allowing files to be renamed or deleted. The motivation for including these was the need to be able to treat a FIAP file as a single entity even in cases where the server actually uses several files (e.g. separate index files).

To be able to perform only limited file management is unsatisfactory. For the sake of consistency, either complete file management should be available (as far as practicable) or file management should be left to a separate file management protocol. The latter course is not, however, terribly realistic; changing protocols in the middle of an application is at best unwieldy, and the file management system would have to know about the implementation of FIAP files used by the file server to continue treating them as a single entity. Future versions of FIAP should, therefore, incorporate more comprehensive file management facilities, allowing the manipulation of sub-directories and access to directory information as appropriate. NFS [Sun86] and ISO FTAM [ISO84] are two protocols that try to address this in a machine-independent way; both provide an idealised filestore (the NFS filestore is based on the Unix filestore, while ISO define their own Virtual Filestore). File management operations on the idealised filestore are then translated into 'best fit' equivalents for the local filestore.

One aspect of both file management and file operation that FIAP does not cater for is file security and access controls. Specifically, while the server is, being a user process, subject to whatever file access restrictions the user is bound by, it in turn imposes some of its own. There is no way at present to request read-only access to a file, or indeed any other form of limited access (e.g. add only, update only). So when the user attempts to use a file on which read permission is available but not write permission the attempt fails because the file server must assume that full access is required, and this is quite properly denied by the mainframe file system. It should also be possible to create files with specific access permissions — the addition of a complete set of file management operations would also include a definition of file access permissions supported and means of changing those applicable to a particular file.

If FIAP is to be useful in environments where many users are accessing and modifying a particular file, some mechanism for locking records and files must be adopted. It is, again, imperative that this is done using local mechanisms if at all possible, so the locks are recognised by mainframe programs. Fortunately

this is one area where the server can adapt limited local capability to a more general case specified by FIAP. For example, if some form of automatic record locking is required, then the server can implement this on a mainframe operating system where all locking is manual i.e. must be requested by a program.

To expand briefly on a possible set of locking facilities for FIAP†. Locking should be available at both file and record level, and locks should be either exclusive, permitting no access to any other process trying to use the file or record, or read, permitting other processes to read but not modify the file or record locked.

Given the slow speed of the connection, and the consequent need to keep the number of FIAP operations that have to be executed by a program to a minimum, automatic record locking should be available. This may be accomplished by placing a lock on a record as it is read, and releasing that lock when attention shifts to another e.g. on any read, write, seek, find etc. to another record (the other record being locked instead as it becomes the focus of interest). The FIAP multiple record operations should probably lock all records read or written by the operation until attention moves on, i.e. treat all affected records as one for locking purposes. Manual locking is simply a matter of specifying that the lock should not be removed until explicitly instructed. It should be possible to request this on a record by record basis.

In addition to new locking operations, existing FIAP operations will also be affected. To be able to use read file locking, it must be possible to specify when opening a file that only read access is required — another reason for adding file access permissions. The response of operations on meeting a lock also needs to be selectable; waiting for the lock to disappear, returning immediately with an error or returning with an error if the lock has not been lifted after a certain time are possibilities.

§1.5 notes that one of the problems with the initial work on extending Kermit to allow file access to a bytestream was the difficulty in adding further extended file operations as required. Adding such operations to a FIAP system is undoubtedly easier; it is not necessary to modify the system on several levels. The extra code required has to handle the transmission of parameters to the server, their reception at the server, the execution of the new operation, and the return of the result. Most of this is simply adding the necessary

† This set of locking facilities is similar to that provided by several systems e.g. VAX/VMS and X/Open CAE.

wrapping to the actual operation code to enable it to be called via a remote procedure call, and can readily be generated by making minor modifications to existing code†; it would not be difficult to generate the required code automatically given a suitable description of the parameters to be passed and results returned.

3.4.2. The FlAP protocol

The central feature of the FlAP protocol is that it deals with only one ongoing file operation at a time. It is not possible to request a file operation if one is currently active. Every operation goes through the same cycle:

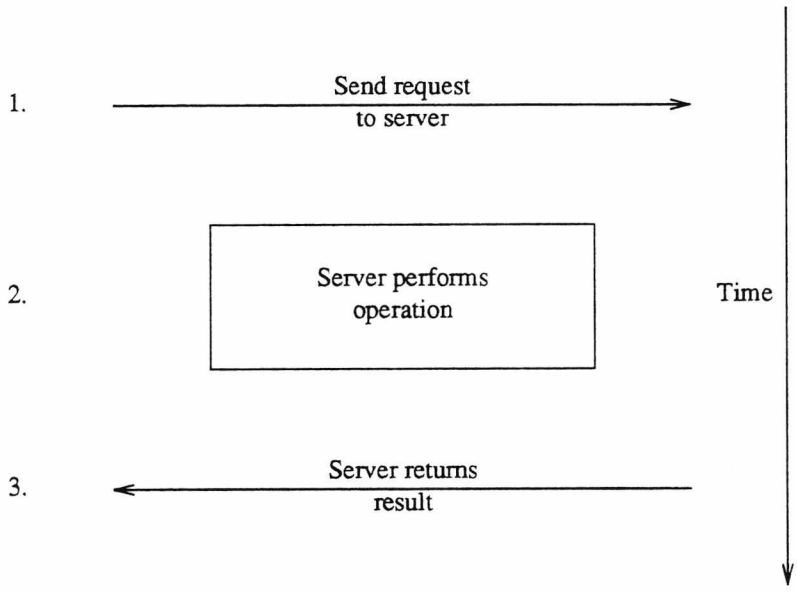

Figure 3.7: FlAP operation cycle

This is analogous to an Idle-RQ protocol. Rather than a 'send a packet, get response, send next packet' cycle, FlAP imposes a 'send a request, get response, send next request' cycle. A greater throughput of file operations could be obtained if a Continuous-RQ approach was adopted, where requests were sent when made (subject to limitations of buffer space and so forth). This is implicitly acknowledged in the provision of multiple record read, add and update operations; these are faster for precisely the reasons discussed above.

† It must be admitted that, although straightforward, writing this is a somewhat tedious job.

However, there are problems with having multiple record operations outstanding at once. Two main issues arise — interface complexity and error handling.

To illustrate the latter first, consider what the response of the server should be when faced with an error arising in the first of a sequence of outstanding operations. For example, suppose the following sequence of operations is pending:

Seek	Read	Seek	Update	Seek	Add
a	b	c	d	e	f

Obviously, if Seek(a) fails, Read(b) will return an incorrect result, and if Seek(c) were to fail Update(d) would overwrite the wrong record. However, with current FIAP semantics, Add(f) would be safe if Seek(e) should fail†. What if Read(b) fails? This may be intended, reading a record to locate the end of the file. So if an operation fails, we may want the server to ignore the failure and carry on, abort all currently outstanding operations until some indication arrives that the application has recognised the failure, or see if the next operation is marked as dependent on the previous one.

This leads back to the question of the complexity of the file system interface. If multiple file operations are to be outstanding at any one time, the results of these functions will become available one by one and must be checked by the application program to confirm that the expected action has taken place. The mechanics of the interface are not too great a problem — for example, the I/O interface to the VAX/VMS operating system readily permits multiple concurrent I/O requests to be issued. Making effective use of it would be difficult to achieve; there are few occasions when it is not necessary to ensure one file operation has succeeded before proceeding with another. Some scheme for categorising operations by what was to happen if they failed could doubtless be devised; however it was felt that the gain in throughput that might be realised was not enough to justify the additional interface complexity and implementation effort.

One possible means of helping FIAP to make best use of the available communications bandwidth would be to employ some form of data compression — if a 200 byte record can be sent compressed in 100 bytes, the effective speed of the link is doubled. The tradeoff here is that sophisticated data compression

† Relative text files are, of course, an exception.

takes time; that time must be balanced against the time savings resulting from having to send less data across the link. It is also a moot point as to whether any data compression is not better done by the transport service. However, data compression may simply not be available from the transport service, so perhaps a data compression mechanism that could be used on FIAP packets before they are passed to the transport service should be defined, to be used as required. This would in turn required the FIAP protocol to be modified, so that compressed packets could be recognised and decompressed before further processing.

Finally, a minor improvement to the handling of vector operations in the current implementation. Ideally, the processing of each record should be done as soon as the appropriate parameters are received. The time taken to process the record can then overlap with the time taken for the next record's parameters to arrive. The current implementation does not dispatch parameter packets until they are filled — when reading records with *flqVGet*, this means that requests for over 100 records can be bundled into one packet (i.e. MMMS message) and are not delivered until the entire packet has reached the server. This means that, especially at low line speeds where the time taken for the parameters to reach the server can be appreciable, greater throughput could be obtained by more intelligent use of packets. If parameters were to be sent in shorter packets, less time would be lost waiting for them to travel to the server. Similar remarks apply in the case of *flqVGet* to record data sent by the server — currently it can all be buffered into one packet at the server which is not sent until the last record has been processed. Transmission of the contents of the first record *could* start as soon as they are available.

3.4.3. FIAP file organisations

One of the aims of FIAP, laid down in §3.2.1.1, is that files created and manipulated by FIAP should use local file organisations as far as possible, so that the files may also be manipulated by mainframe programs. Using local file organisations, however, only addresses half the latter goal. The data inside the records must also be meaningful to the mainframe program.

When considering text files, one answer would be to stipulate that all communication should take place using one selected character coding. Translation (e.g. from EBCDIC to ASCII) would be handled by the file server.

Of course, text does not only occur in text files. It is also not the only data representation possibly requiring translation. Integer values over one octet long may be stored either low octet first or high octet first. Floating-point values may be represented using a variety of different formats (even on one machine), of varying degrees of precision. On a higher level, a standard representation of date and time might be desirable.

A solution would be to specify not the record length but the composition of the record when creating the file. The ISO Virtual Filestore takes this approach, building up files as a combination of File Access Data Units (FADUs), each of which is in turn composed of Data Units (DUs).

Similar considerations apply within text records. Consider a text file which attempts some simple formatting using a form-feed character at the top of each page, and tab characters with the implicit assumption that tab stops are set every eight characters. Should a conversion to a suitable local convention be attempted? Also, §3.2.2 noted that while it is possible in the current implementation to read a text file which does not have a newline character as its last character, it is not possible to write one. The nub of the problem is that Unix allows lines of text to be terminated by the end of the file as well as by newline characters. If a record terminated in such a manner can be written, what should be the proper behaviour when adding another record to the file?

The concessions FIAP makes to the mainframe environment are currently concentrated on making it possible to read records using local facilities, as far as practicable. It makes no attempt to alter the record to conform to local conventions; the above few paragraphs give some idea of the problems involved. A case can be made for requiring character set translation in text files where appropriate, as it is a trivial thing to do.

FIAP indexed files only support one type of key, and one collation order — keys must be made up of sequences of characters and indexes must always be held in ascending order. This was specified in an attempt to make the capabilities a subset of those provided by typical indexed file facilities. With hindsight, allowing keys to be made up from several sections of a record is questionable on those grounds.

Possible additional facilities include indexes sorted in descending order, keys made up from numeric or other data types not based on characters and some form of key compression. There is a balance to be

struck here between convenience for the application programmer, use of local file organisations and the effort needed to implement a FIAP file server. This latter consideration is especially germane when considering indexed files, as they are not at all trivial to implement from scratch.

Finally, one small but important omission from the current FIAP specification is minimum requirements for various limits within the system. The current implementation imposes an overall limit on record length, and restricts the number of keys per indexed file to 8, the number of segments used to make up each key to 8, and the maximum length of a key to 80 characters if duplicate keys are not allowed in the index, or 77 characters if they are. The intention behind setting minimum values for the above would not be to forbid implementations from exceeding them, but to provide a minimum standard service for applications likely to be used with more than one FIAP file server.

Chapter Four

Applications

To demonstrate that using FlAP is feasible in application programs, two simple application programs have been written. The first is a customer records file maintenance program, the second a simple screen editor. This chapter gives a brief description of them, the FlAP functions they employ and their response time in use. It concludes with an examination of the problems likely to be encountered when adapting existing programs to use FlAP.

Both application programs run on IBM PCs or equivalent, using the FlAP library and Unix FlAP server described above. When started from a Unix command line, both take an optional *-d*, *-dd* etc. parameter, which records FlAP server debugging information. See the UNIX-SERVER entry in the FlAP Programmer's Manual for details.

4.1. Customer file maintenance

This application manipulates records holding details of customers as might be required by a small company. Records may be added, updated or deleted to/from a customer file held on the mainframe. The file itself is a fixed-length record indexed file. The data is stored one record per customer, and contains a customer code, the customer name and address (including postcode), the names of up to two contacts at the customer, a credit code and up to three lines of comment. The total record length is 379 bytes.

4.1.1. Operation

The program is started from the Unix command line using the protocol outlined in §3.3.3. Optional parameters are enclosed in [..].

```
% customer [-d...]
```

Once started, the program establishes contact with the FlAP server and opens the customer file. If the file is not present, it is created. The main screen is displayed, which includes a count of the number of records currently in the file, and the cursor placed in the customer code field.

The user may then select a record in one of two ways. The first is to enter a customer code (blank

codes are not permitted), which will either retrieve the corresponding record or, if no such record is found, prepare for entry of a new record. The alternative is to read either the previous record or the next record using the ↑ key to select the previous record or the ↓ key to select the next record. Record selection 'wraps round'; i.e. if the next record is requested at the end of the file, the first record in the file is read. Once a record is selected, the cursor moves to the first data field (the customer name) and the record may be edited. When editing is completed, the user may save the changes, return to the editing stage, cancel the update or delete the record. In the case of the delete option, additional confirmation is requested before the record is deleted. Should a record actually be added, updated or deleted the appropriate action is requested of the FIAP server, the screen cleared and the user prompted again for a customer code. Pressing ESC instead of entering customer code exits the program.

When entering data into a field, several editing functions are available. These are as follows:

←	Move one place left
→	Move one place right
HOME	Move to the start of the field
END	Move to the end of the field
Backspace	Delete the previous character
DEL	Delete the current character
INS	Insert a space before the current character
↓ or Return	Move to the next field
↑	Move to the previous field
PgUp	Move to the first field on the screen
PgDn	Accept all the fields on the screen
ESC	Abandon the current screen

Table 4.1: Customer data entry editing functions

4.1.2. File access functions and methods used

For the most part synchronous FlAP functions are used, as there is only limited scope for making profitable use of asynchronous functions. In particular, once a record has been requested there is little that can be done other than to wait for it to be read.

The first thing to be done when the application starts is to establish contact with the FlAP server, open (or possibly create) the customer file and find the number of records in the file so that this can be displayed. The following FlAP functions are used:

flInit	Initialise interface
flStart	Try to establish contact with FlAP server
flReady	See if contact established
flOpen	Open customer file
flCreate	Create customer file if not found
flSeek	Return number of records in file

Table 4.2: FlAP functions used in customer initialisation

The user may then request a record be retrieved, either by entering a customer code or requesting the previous or next record. The following functions are used:

flFindRec	Retrieve record using customer code
flGetRec	Read next record
flVGet	Move to and read previous record

Table 4.3: FlAP functions used in record retrieval

The record contents may then be edited, and the record either added as a new record, rewritten or deleted. Each of these can be done asynchronously while the next customer code is being entered.

flqDelRec	Delete record
flqVAdd	Add new record
flqVUpdate	Rewrite existing record

Table 4.4: FlAP functions updating records

A vector operation is not strictly needed when adding records (where the record is added in the file is independent of the file currency), but is used as a matter of convenience — the function concerned with updating a record prepares the vector and selects *flqVAdd* or *flqVUpdate* at the last moment. A vector operation is used on rewriting because currency must be set to the record to be rewritten.

The progress of the asynchronous operations must be monitored. This is done in the simplest possible way — before any FlAP function is used, the application waits, if necessary, for the previous one to finish.

flDone	See if asynchronous operation finished
flWait	Wait for asynchronous operation to finish

Table 4.5: Monitoring FlAP asynchronous operations

Finally, when the user exits the application, the customer file must be closed and the FlAP server shut down.

flClose	Close customer file
flShutdown	Shut down the FlAP server

Table 4.6: Functions used tidying up before exiting

4.1.3. Responsiveness in use

Some measurements of the time taken by the application to perform various tasks are presented below. Times are given over a range of line speeds for the following:

Startup The time between issuing the command to the Unix system and the application being ready to accept the first customer code. This covers the time taken for the application starter protocol to exchange program name and parameters, the loading of the application, making contact with the server and opening the customer file.

Find+Read Time taken for an existing customer record to be located using the customer code and read. The record used had entries in all fields — records with empty fields take less time to read, as they benefit dramatically from MMMS's run-length compression.

Find Fail Time taken to determine that a customer code does not already exist and presumably heralds the entry of a new record.

Write Time taken for a record (the same record used in *Find+Read*) to be written. Of course, this takes place while the next customer code is being entered, so a sufficient delay there will allow writing to complete without causing any apparent delay.

Read Prev. This and *Read Next* give the time taken to read the preceding and next records respectively. In neither case did any 'wrapping around' from first to last record or vice-versa take place.

Line speeds are given in characters per second. Times were taken by hand, and rounded to the nearest half second. At the time the measurements were taken, the Unix system hosting the FIAP server was servicing two other users.

Speed	Startup	Find+ Read	Find Fail	Write	Read Prev.	Read Next
960	5.0	1.5	1.0	1.5	1.5	1.0
480	4.5	2.0	1.0	2.0	1.5	1.5
240	6.0	2.0	1.0	2.5	2.0	2.0
120	6.0	3.5	1.5	4.0	3.5	3.5
30	12.0	12.0	2.5	12.0	11.5	12.0

Table 4.7: Customer application response times

Some of the times given appear to diverge from the expected, notably the *Startup* time at 480 cps. This, in fact, reflects the effect of variable response times from the mainframe due to local load fluctuations. The *Startup* time at 480 cps over a series of readings is usually between 5.0 and 5.5 seconds.

4.2. RBed

The second FlAP application is RBed†, a simple screen editor that manipulates text files held on the mainframe. The editing facilities provided by RBed are by no means comprehensive — there is no search and replace facility, for example — but suffice for text entry and minor changes. All screen handling and modification within a line is handled by the PC, while the FlAP server updates file contents.

RBed suffers from several internal limits imposed in the name of simplicity of implementation, notably a restriction on the maximum line length of 150 characters (excess characters are silently ignored and may be lost if the line is changed) and a maximum of 65535 lines in the file. There is no overall limit on filesize except that imposed by these two restrictions.

4.2.1. Operation

RBed is started from the Unix command line using the protocol outlined in §3.3.3. The form of the command is as below — optional parameters are enclosed in [..]. A filename must be supplied.

% RBed [-d...] *filename*

Once RBed is running on the PC, it clears the screen, writes its header line and establishes contact with the FlAP server. The mainframe file is prepared for editing (see below, §4.2.2) and a number of text lines are read into the PC and displayed ready for editing. The PC screen display is as overleaf.

The status line gives the name of the file being edited, the current position of the cursor and the current text addition mode. This is either *Insert*, in which case any text typed is inserted before the current cursor position with any existing text moving right to make room, or *Overwrite*, where text typed simply replaces any existing text.

† RBed = **R**avenous **B**ugblatter **ed**itor. "A beast so mind-bogglingly stupid ..." [Adams85].

```
    RBed v1.00        Editing two.txt              Insert   Line   24 Col  66
.FS \(dg
RBed = \fBR\fRavenous \fBB\fRugblatter \fBed\fRitor. "A beast so
mind-bogglingly stupid ..." [HHGTG].
.FE
that manipulates text files held on the mainframe. The editing
facilities provided by RBed are by no means comprehensive \(em there
is no search and replace facility, for example \(em but suffice for
text entry and minor changes. All screen handling and
modification within a line is handled by the PC, while the FlAP
server updates file contents.
.PP
RBed suffers from several internal limits imposed in the name of simplicity of $
maximum line length of 150 characters (excess characters are
silently ignored and may be lost of the line is changed) and a
maximum of 65535 lines in the file. There is no overall limit on
filesize except that imposed by these two restrictions.
.NH 3
Use
.PP
RBed is started from the Unix command line using the protocol
outlined in \(sc3.3.3. The form of the command is as below \(em
optional parameters are enclosed in [..].
.DS B
```

Figure 4.1: Sample RBed screen

Any text line longer than 80 characters (the width of the PC screen) is indicated by a '$' in the final position on the line. If the cursor is placed on the line and moved rightwards, the portion of the line displayed changes to ensure the cursor position is always on the screen.

The following functions are available within RBed.

PgUp	Move up a screenful
PgDn	Move down a screenful
Control-HOME	Move to the start of the file
Control-END	Move to the end of the file

Table 4.8: Continued ...

↑	Move the cursor to the same column on the previous line
↓	Move the cursor to the same column on the next line
←	Move the cursor left one position
→	Move the cursor right one position
Control-←	Move to the start of the previous word
Control-→	Move to the start of the next word
HOME	Move to the start of the current line
END	Move to the end of the current line
INS	Insert a space at the current position
DEL	Delete the current character
Backspace	Delete the previous character
f1	Display a summary of commands
f2	Toggle *Insert/Overwrite* modes
f3	Insert a new blank line
f4	Delete the current line
f5	Search for a pattern
f10	Save changes and exit
F10	(Shift-f10) Quit without saving changes

Table 4.8: RBed edit commands and keys

In addition, if the -d parameter was given Alt-f1 prints some information on RBed internal data structures.

Backspacing past the start of a line causes that line to be deleted and its contents appended to those of the previous line. A RETURN causes the current line to be split at the current cursor position, and all line contents after the cursor to be moved onto a new line.

When a search is requested, RBed prompts for a Unix-style regular expression and searches for characters matching that pattern, placing the cursor at the start of the pattern if found. The current cursor

position is unchanged if the search fails. Searching proceeds from the current position to the end of the file, and may be interrupted with the ESC key.

4.2.2. Implementation

4.2.2.1. Overview

Once communication is established with the FIAP server, the file to be edited is copied to a temporary relative text work file. Changes are made to this work file and on exiting the original file is renamed† to preserve a backup copy of the file before changes were made, and the work file is copied back to a sequential text file with the original file name (assuming changes are to be saved).

The RBed internal data structure holds lines in groups, or *chunks*. Chunks may be either *empty*, in which case the line contents are not currently in memory, or *full*, in which case the chunk maintains the line contents in a linked list. To illustrate, on initialisation the data structure is as below; one empty chunk representing the whole file of *n* lines.

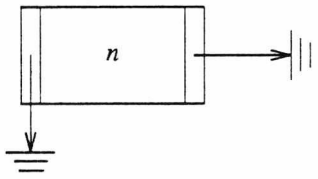

Figure 4.2: RBed internal structure — empty

Line contents are read in as required, by default 50 at a time, to form new full chunks. So, once the first batch of lines has been read in so the first screen can be displayed the data structure is as below.

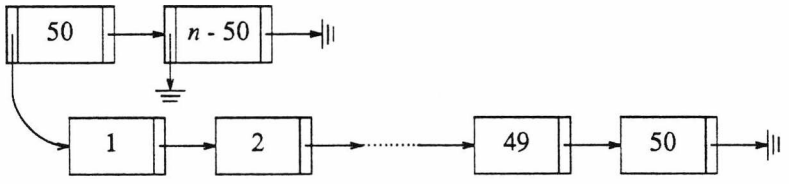

Figure 4.3: RBed internal structure — 50 lines present

† *.bak* is appended to the original name. For example, saving changes to a file named *zaphod* causes the original file to be renamed *zaphod.bak* and a new changed file *zaphod* to be created.

Chunk and line buffers are drawn from a pool of each when required, and returned to the pool when freed (adjacent empty chunks, for example, are automatically merged; adjacent full chunks may be merged if their combined size is not excessive). When the supply of line buffers is exhausted, the full chunk which has been the least recently displayed is converted to an empty chunk and the line buffers reclaimed.

Each line carries two flags indicating whether the line has been altered since it was last read in (or updated), and whether it is a new line (one that is not yet present in the work file). RBed maintains a notion of the current line, and when that line is vacated (i.e. another line becomes the current line), it is examined to see if it needs to be added to the work file or rewritten, and appropriate action is taken.

4.2.2.2. File access functions and methods used

As with the customer records program, RBed mostly uses synchronous FlAP functions. However, it does use asynchronous functions to update the work file, and this enables it to support continuous text entry on the slowest of lines.

Again, the first thing to be done when RBed starts is to establish contact with the FlAP server. The file to be edited is then opened, the work file created and the file contents copied to the work file. The result of the copy gives the number of lines in the file, and enables the internal data structure to be initialised. If no file of the given name is found, an empty work file and internal data structure are created.

flInit	Initialise interface
flStart	Try to establish contact with FlAP server
flReady	See if contact has been established
flOpen	Open the file to be edited
flCreate	Create the work file
flCopy	Copy file to work file
flClose	Close original file

Table 4.9: FlAP functions used in RBed initialisation

All access to lines in the file is done via one function, which checks to see if the desired line is present in the internal data structure and reads it in if not. If a line has to be read in, an entire chunk of up to 50 lines is read with the required line as near to the middle of the chunk as possible.

The previous section noted that lines are updated when they cease to be the current line. Possible updating actions are to add the line as a new line, to rewrite it or to delete it, and each is done using one or more asynchronous operation. Since the next FIAP operation will not be required until either a changed (or new) line is vacated, or a line not currently held internally must be read, in the majority of cases the time elapsing between FIAP operations allows asynchronous updating to complete without delaying the user.

Two operations are required when deleting a line, to move to the correct record and then to delete it. When reading, adding or updating lines, vector operations are used which combine location of the desired record with updating.

flVGet	Read lines
fiqVAdd	Add a new line
fiqVUpdate	Rewrite a line
fiqSeek	Move to line to delete
fiqDelRec	Delete line
flDone	Monitor progress

Table 4.10: FIAP functions reading and modifying lines

Progress is monitored during text entry, and any results and/or error codes compared to those expected. Unexpected results cause a fatal error. If a line deletion is in progress, a flag indicates this and causes the monitoring routine to issue a *fiqDelRec* when the *fiqSeek* completes successfully.

Searching for a pattern causes two searches to be started simultaneously. A search request is dispatched (asynchronously) to the FIAP server and meanwhile the current line and any subsequent lines in memory are searched until either the search succeeds or a gap in the sequence of lines in memory is found (or the end of the file is encountered). If the search of the internal data structure succeeds, the search request

made of the server is aborted; otherwise, if the internal search fails, RBed waits for the server search to finish and, if the pattern is found, moves to the line in question — this may well require a further chunk of lines to be read in.

flqSearch	Search for a pattern
flqAbort	Abort search

Table 4.11: FlAP functions used searching

The user will usually require any changes made to be saved on exit, but may request that they be abandoned and the original file be left untouched. If changes are to be saved, the original file is renamed to become a backup copy, and the work file contents copied to a newly-created file with the original file name. In either case, the work file is closed (and thus automatically deleted) and the server shut down before FlAP exits.

flRenFile	Rename original file
flCreate	Create new file
flCopy	Copy changed text to new file
flClose	Close new and work files
flShutdown	Shut down FlAP server

Table 4.12: FlAP functions used on exiting

4.2.3. Timings and impressions of use

The table below gives some idea of how quickly RBed performs certain actions at various line speeds. The file being edited was 173 lines long and contained 3889 characters. The chunk of lines read in comprised 50 lines and 1394 characters. Times are given for the following actions:

Startup The time elapsed between issuing the command to start the editor and the start of reading in the first chunk of lines for display. This covers starting RBed, making contact with the

FlAP server and copying the file contents to the work file.

Read The time taken to read this first chunk of lines. Adding this to *Startup* gives the elapsed time between issuing the command and being able to edit the first line.

Write Line How long it takes to rewrite a line of 75 characters. Normally this takes place while the next line is being edited, and so causes the user no delay.

New Line The time taken to add a new empty line. Again, usually done while the next line is being edited.

Empty Startup The time elapsed between issuing a command to edit a new (i.e. non-existent) file and being able to start entering text. This covers starting RBed, making contact with the server and creating the work file.

As with the customer file program, times were taken by hand and are given to the nearest half-second. Line speeds are in characters per second. At the time the measurements were taken, the Unix system hosting the FlAP server was servicing seven other users.

Speed	Startup	Read	Write Line	New Line	Empty Startup
960	12.5	8.5	1.0	1.0	5.0
480	10.5	8.0	1.5	1.0	5.0
240	12.0	14.0	1.5	1.0	5.5
120	12.0	26.0	2.0	1.0	6.0
30	20.0	95.5	5.0	2.0	11.0

Table 4.13: RBed response times

By way of comparison, the next table gives startup and screen redrawing times for two Unix-based screen editors, *vi* and a version of MicroEmacs (see §4.3 below for more on MicroEmacs) editing the same file used in the previous timings.

	vi		MicroEmacs	
Speed	Startup	Redraw	Startup	Redraw
960	3.0	3.5	1.5	3.5
480	3.0	3.5	1.5	3.5
240	3.0	3.5	1.5	4.5
120	3.0	7.5	2.0	10.0
30	4.0	28.0	3.5	37.0

Table 4.14: *vi* and MicroEmacs response times

The timings were done on the same equipment used for the RBed timings. The terminal emulator used was the VT100 emulator described in Appendix A, which uses the same serial port handler as MMMS. Faster microcomputers and/or faster terminal software improve screen updating times at faster line speeds. A screen redraw in both cases rewrites 23 lines, just under half the number of lines read by RBed at once.

In use, the most noticeable difference between RBed and the Unix-based editors is the time that elapses on startup before being able to commence editing. At a line speed of 240 cps this is 26 seconds for RBed compared to 6-7 seconds for the other two.

During editing, the delays caused by reading lines are initially obtrusive. However, as editing progresses, the proportion of the file held internally by RBed increases and consequently delays become less and less frequent (when run on a PC equipped with the full complement of memory, RBed can hold around 2000 lines internally). At slower line speeds the benefits become increasingly apparent, as less time is wasted on screen updating. In particular, the RBed screen always reflects the current file contents, while at lower line speeds a screen redraw can mean the changes made are not reflected until the redraw has finished.

In summary, RBed seems to the writer† to offer advantages over the more conventional editors for non-trivial editing sessions at lower line speeds. Initially, delays are longer, but they are more concentrated.

† Not, unfortunately, a totally disinterested party.

The user may be idle for 30 seconds or more, but is not subject to smaller but continual delays.

The advantages decrease at higher line speeds as the conventional editors become more usable. RBed still suffers from a slow startup (mostly due to the overhead of the microcomputer having to load the RBed program, and then copy the file to the work file), and it is difficult to see what can be done to much improve this. The disappointing throughputs achieved by MMMS at higher line speeds also play a part — reading of lines in particular takes longer than it might.

4.2.4. Comments and improvements

In the opinion of the writer RBed succeeds in its aim of demonstrating that text files may be edited by a program accessing files only through the FIAP server. However a few simple enhancements would greatly improve its usability.

The first, and most important, is some means of guarding against the possibility of data loss due to circumstances beyond the user's control. The means can range from simply providing a means of saving changes made without having to quit the editor, through doing such saving automatically (say after a certain number of keystrokes) up to allowing interrupted editing sessions to be resumed. This latter could be accomplished by not deleting the work file on a failure exit, but providing an option requesting the editor to scan for work files and recover their contents.

In normal use RBed responds to user requests in a reasonable time. However, when making requests of the FIAP server, delays can occur if the operation takes significant time (e.g. copying to or from the work file) or the communications link fails and the user has to wait for it to time out. There should be a means of aborting any operation and exiting RBed swiftly†. This applies equally to the customer records program.

The list of additional facilities that could be added to RBed is long. A few examples are importing text from other files, block move/copy/delete, word-wrap and delete to end of line. All these could be added reasonably easily. Other operations affecting many lines in the file but not directly translatable to single FIAP operations, such as search and replace, can be implemented but will suffer from the communications

† RBed could be further integrated into the local Unix environment if a means of suspending an editing job for later resumption existed. Providing such a facility would be a complex matter.

bottleneck. To use the example of search and replace, changing 300 occurrences of, say, 'Xmodem' to 'Kermit' requires each line to be changed to travel from server to PC, be modified and then sent back again. If the average line length is 30 characters, this could take up to 75 seconds on a 240 cps link.

Reductions in the delays experienced by the user are dependent on improvements in FIAP and MMMS, in particular improvements in MMMS throughput at higher line speeds. Also, the time taken to read lines at slow line speeds bears out the point made above (§3.4.3) about adjusting size of FIAP packets. Of the time of 95 seconds to read lines at 30 cps, nearly 20 seconds is taken up with transmitting the request to the FIAP server. If the necessary data for the first line to be read was sent as a separate packet, the server could being returning line contents while the rest of the requests for lines were in transit.

4.3. Conversion of existing programs to using FIAP

This section examines the difficulties faced when adapting existing programs to use FIAP by considering the problems posed by one particular program, MicroEmacs [Lawrence87]. It should be stressed that a conversion has not actually been carried out, only an assessment of the potential problems and possible solutions.

MicroEmacs is a public domain text editor with a user interface patterned after the popular EMACS editor [Stallman81]. It is written in C and is available for a wide range of systems from IBM PC series microcomputers to DEC VAX mainframes. It was selected for analysis because it is a non-trivial program for which source code is widely distributed.

4.3.1. MicroEmacs internals

MicroEmacs edits text held in one or more buffers, one buffer per file. Each buffer in turn heads a list of line structures holding the text for each line. Files are read into a buffer before being edited, and written out when editing finishes. MicroEmacs holds the entire file in this internal structure during editing.

Two placeholders are maintained within the buffer, *dot* and *mark*. *dot* is the current position within the buffer and is altered by nearly every editing action. *mark* is set explicitly by the user, and with *dot* delimits a block of text which may then be manipulated by various block commands.

MicroEmacs provides many editing functions (over 120 in the latest version) which range from cursor movement to searching. Commands are 'bound' to keystrokes or keystroke sequences. A default set of bindings is provided, but users may alter bindings to reflect personal preferences. Some of the more complex and/or rarely-used commands have no default binding, and must be executed by the *execute-named-command* function.

MicroEmacs source code is split into several modules, each dealing with various aspects of the editor. For example, all interaction with the file system is confined to one module, as is all interaction with the screen and keyboard, so porting MicroEmacs to a new system usually involves little more than writing appropriate code for these two modules. It is not only system dependent aspects of the editor that are separated into modules; all manipulation of line contents is confined to one module, while all functions that alter *dot* are contained in another module.

4.3.2. Conversion problems

A trivial conversion to FlAP could be accomplished simply by providing appropriate routines in the file access module to read the whole file in and write the whole file out. While acceptable when editing small files, the delays incurred when reading and writing large files would not be acceptable. A realistic conversion to use FlAP must remove the requirement that the whole of the file being edited be held in the internal buffer structure. This means it is no longer necessary to read lines other than those required, but shifts the location of the edited copy of the file out of the internal buffer structure to a file, so changes to the buffer must be reflected in the file.

This suggests that MicroEmacs must be modified to use a temporary relative text work file in the manner of RBed, copying the file to be edited to the temporary file, making changes and then copying back. So the basic changes to be made to MicroEmacs are to modify the internal buffer structure to allow for unread lines and read them in when required, and to rewrite changed lines when attention passes to another line.

Borrowing from experience with RBed, the line structure needs flags added to indicate if the line has been read, if it has been modified and if it is a new line. Setting the latter two would not be a problem —

MicroEmacs already has functions called whenever a change is made and whenever a new line is required.

All movement between lines in the buffer data structure is done using functions that move to the next or the previous line, which would need to be modified to read in lines not present.

Writing changed lines depends on detecting when *dot* moves off the current line. As already noted, all functions that move *dot* are collected into one module, so this does not appear the be too difficult. The chief problem here is line numbers. MicroEmacs does not maintain the number of the current line within the file, requiring either that currency in the temporary file track *dot* as it moves between lines, or that line numbers are available when required. As the former option requires a file operation every time *dot* moves between lines (even if the line being vacated was unaltered), the second approach is greatly preferable. The simplest solution would be to add line number information to *dot* and *mark*.

Once the above changes have been made a few areas of difficulty remain. These are operations that, although they would function correctly, need further work to make best use of FlAP functions.

Searching is the most obvious. MicroEmacs allows buffers to be searched for a pattern, which is by default searched for as-is, but may optionally be a Unix-style regular expression. To avoid reading in the whole file on an unsuccessful search, searching must be at least partially carried out by the FlAP server (as in RBed, lines may be searched in the internal data structure until a line not yet read in is encountered). In the default case, where no special characters are to be recognised, the pattern will require some preprocessing to ensure any such characters are matched correctly before it may be handed to the FlAP server.

More insidious problems are posed by the KILL buffer. MicroEmacs places all deleted text into this special buffer, from whence it may be recovered. It is also possible to copy blocks of text to the KILL buffer. While all additions to the KILL buffer are handled by one routine, this routine simply deletes a given number of characters from before or after *dot*. This causes problems if not all the text within the region is in fact held in the data structure, as it must be read in to discover how many lines must be deleted — also, deletion of large regions of text would be best handled by copying them to a KILL file, so that recovery of that text would not involve transferring it all across the link. As additions to the KILL buffer are always takes from the region between *dot* and *mark*, or are as the result of a request to delete *n* lines starting from *dot*, then given the alterations detailed above the line numbers involved would be available.

This does not, however, make the necessary alterations trivial.

There remain areas which cannot be satisfactorily dealt with. Search and replace, for example, involves fetching each line containing the pattern to be replaced and rewriting the updated line. So replacing a frequently-occurring pattern in a large file would be time-consuming. MicroEmacs also offers functions that change the case of characters between *mark* and *dot*, which would again be time-consuming over large regions for the same reason. These are functions not provided by the FlAP server, and so cannot be speeded up without adding custom code to the server. Finally, some versions of MicroEmacs give the user access to the underlying operating system by either allowing themselves to be suspended or by starting a new command line interpreter. Again, these are not functions supported by FlAP.

To summarise, conversion of MicroEmacs to use FlAP is possible. It is, however, by no means trivial, requiring changes outside the scope of those considered normal when porting programs. Omission of a few functions and slow operation of others appears inevitable, but not to the extent where the suitability of the editor for everyday use is unduly impaired.

The problems encountered are, in the main, caused by the assumption that file operations do not take long to complete. In the case of MicroEmacs, the prime manifestation of this is the wholesale reading and writing of the file being edited to/from the internal buffer.

Chapter Five

Conclusions

This last chapter starts with a summary of the previous chapters, followed by an assessment of what has been achieved. It gives some suggestions for future work and final conclusions.

5.1. Summary

This thesis started by outlining the benefits to be drawn from a microcomputer-mainframe link, and looked at the various connection mechanisms available and ways of using these links. Currently, terminal lines are the predominant means of connection and will probably remain important for some time to come (see below, §5.3), so this work has concentrated on ways of using them. Four methods of interaction were identified, namely

— Terminal emulation

— File transfer

— Server

— Integrated application

Integrated applications can make the best use of terminal lines, as they can minimise the volume of data that has to be transferred. They can additionally employ sophisticated user interfaces impossible on a conventional terminal, and also reduce the load on the mainframe. They are, however, difficult to construct; two programs must be written and they must interact correctly.

The range of applications under consideration was then narrowed to those using the mainframe only to access files. It was suggested that the benefits of such integrated applications could be realised by a simpler server mechanism, with a file server on the mainframe providing a defined service to the application. File servers have long been used over high-speed links — however, the low speed of the link would seem to preclude their use over terminal lines. However, it was hoped that a combination of the following methods would alleviate this bottleneck:

— Extended file operations (common file operations, such as copying, performed in the server)

— A record-based file model

— Asynchronous file operations

An outline was given for a system to be built to investigate whether this approach was in fact practical.

The lowest level called for a transport service providing a reliable two-way flow of data between application and server. A survey was given of protocols, mostly file transfer protocols, currently used by microcomputers over asynchronous lines (the most common form of terminal line). None matched the requirement at hand, so a new protocol, MMMS, was described. Details were given of this protocol, its current implementation and its performance.

The next level requirement was for a file access system, comprising a file server on the mainframe and a co-operating microcomputer interface library. Following a discussion on the reasons for preferring a record-based file model over a bytestream model and an outline of the file facilities provided by four record-based file systems, a file access system FIAP was described, which incorporates the ideas outlined above. The description included the file types and operations provided, the protocol used between application interface and server, and a 'starter' protocol for initiating FIAP applications from the mainframe command line.

Finally, two sample (and simple) FIAP applications, a customer records program and a screen editor, were described. Response times for various aspects of their operation were given, and in the case of the editor comparisons made with conventional mainframe-based editors. This was followed by a discussion of the problems likely to be encountered converting existing programs to using FIAP.

5.2. Assessment

The time has come to address the two fundamental questions that this work set out to answer:

(i) Given a slow link, is it possible to build integrated application style programs using just a file server?

(ii) If so, is it easier than constructing equivalent integrated applications?

Briefly, in the opinion of the writer, the answers are a qualified yes and yes respectively.

Turning first to point (i), the existing applications — simple and restricted as they are — do work. So it may be tentatively concluded that applications engaged in file manipulation where the user's attention is focused on one part of the file at a time can be successfully implemented using FlAP. This is, of course, only one category of integrated application, but nevertheless an important one. There is a sense in which the file server capabilities provided could be said to be tailored to the two sample applications; in particular, it must be admitted that implementing an editor without the use of an indexed (i.e. relative text) workfile and the search, copy and duplicate† extended operations would be difficult. On the other hand, an editor built from the start as an integrated application would surely divide the workload between microcomputer and mainframe in a different manner, in particular dealing in text quantities less than a line e.g. deleting one character from a line rather than rewriting the entire line.

As for point (ii), the major difference between a FlAP application and an integrated application is, simply, that only one program has to be written. Life is by no means all roses; more care has to be taken planning file operations than when using a faster file system, and consequently program code can be more complex — experience with the current applications indicates that, while the amount of code devoted to file system interaction is comparable to a conventional program, the additional program complexity means that more work is required overall.

5.3. Future directions

Before suggesting any directions for future work, the future of microcomputer-mainframe links, and the speeds thereof, must be considered. The point of this work has been to attempt to help make use of them despite their slow speed. There would seem little point in carrying out further work if the low-speed links were on the point of being totally replaced by high-speed links.

Chapter 1 mentioned the increasing use of high-speed local area network connections between microcomputers and mainframes. A recent survey [Hodges88] reports that of the 30% of responding organisations that have installed a local area network, only 30% of these LANs are connected to a corporate main-

† Although not currently used in RBed, the *duplicate records* operation would be needed if the editor was to handle blocks of text.

frame. More significantly, over half those who have not installed a LAN indicated that they had no plans to do so in the next 5 years. So it seems that terminal line connection will continue to be of importance.

Furthermore, there is dial-up access over the PSTN. This has, of late, been an area of intense activity, with manufacturers pursuing higher and higher data throughput levels. Appendix G summarises the current state of play — briefly, speeds of over 12,000 bps are currently on offer, though in many (but not all) cases this is achieved partly with the help of data compression techniques. For the future, speeds will continue to increase, but will not rise much above 20,000 bps (without data compression), as higher speeds are simply not attainable over the voice telephone network (see the calculation of Shannon's limit for telephone lines in [Martin76]).

Various suggestions for improvements to MMMS and FIAP are given in §2.5 and §3.4 respectively. These improvements are to the systems as they stand; other possibilities for further investigation are:

— *Analysis of potential extended operations.* As noted in §1.5, the extended operations currently provided by FIAP were selected on an uncomfortably *ad hoc* basis. A systematic analysis of how real-life programs organise and perform file access could suggest other suitable operations (and, indeed, highlight unsuitable ones).

— *Application-programmable extended operations.* Applications are currently limited to the extended operations provided by the file server. If they require some operation not already present, code must be added to the file server. Another approach would be to make the file server programmable by the application, perhaps using some form of file manipulation language. This would allow applications to add functionality to the server as they require, and avoid the danger of the file server continually expanding as new applications appear.

— *Assistance for extended file operations at lower level.* Normal mainframe-resident programs might benefit from the provision of a set of extended file operations. This would be particularly true if support for extended operations, possibly even including hardware assistance†, made them faster than straightforward implementations within the application.

† C.f. the ICL Contents Addressable File Store (CAFS) which uses special hardware to search disc files [Maller79]. CAFS searches can use quite complex criteria; for example, searching a telephone directory for all subscribers with surname 'Bowick' or 'Brown' living in 'Welch Road' in either Chichester or Southsea.

5.4. Conclusions

The system described attempts to alleviate restrictions on file access imposed by slow links with three mechanisms:

(i) Extended file operations

(ii) Record-based access

(iii) Asychronous file operations

(i) and (ii) contribute by trying to minimise the volume of traffic across the link. (ii) helps additionally by making possible the provision of file organisations that reduce the number of file operations required for a particular task (e.g. indexed files). Lastly, (iii) ensures that file access can proceed in parallel with other activities, making maximum use of the available bandwidth.

The class of applications that may make use of this approach is limited, but is none the less useful. The system described presents substantial room for improvement and expansion, the sample applications particularly being comparatively crude. Nevertheless, they do demonstrate the principle. As high-speed local area networks connecting microcomputers to mainframes become more widespread the need for such techniques will diminish (although record based access will still be necessary to make the most of connections to mainframes with record based file systems). However, there remain dial-up connections and terminal links over long-haul networks.

In conclusion, an observation from [Madron87]:

> Is it possible to build an effective, user-friendly micro-to-mainframe connection? Maybe. Is it necessary to build a connection useful to end-users? Without question.

References

Several of the protocol descriptions below have never been formally published. They are available from a variety of sources, including many Bulletin Board Systems, usually under the given **filename**.

[Adams85] Adams, D., *The Hitch-Hiker's Guide to the Galaxy: The original radio scripts*, Pan Books, London, (1985)

[Alisa87] Alisa Systems Inc., *AlisaTalk — AppleTalk Networking for VAX/VMS*. Alisa Systems Inc., Pasadena, California, (April 1987). Promotional material, from JPY Associates Ltd., New Malden, Surrey.

[Anada84] Anada, A. and Marsden, B. "A network operating system for microcomputers" *Computer Communication* **7**, 2, (April 1984), pp 65-72

[Arnold84] Arnold, K. *Screen updating and cursor movement optimisation: A library package* from *Unix Programmers Manual — Supplementary Documents (4.2 Berkeley Distribution)*, University of California, Berkeley, California, (1984)

[Borland85] Borland International Inc. *Turbo Database Toolbox*, Borland International Inc., Scotts Valley, California, (1985)

[Boswell86] Boswell, P. *WXmodem file transfer protocol.* **WXMODEM.DOC**, (June 1986)

[Byrns85] Byrns, J. *Xmodem/CRC overview.* **XMODEM-CRC.DOC**, (January 1985)

[Cargill85] Cargill, T. "Implementation of the Blit Debugger" *Software — Practice and Experience* **15**, 2, (February 1985), pp 153-168

[CCITT84] CCITT, *Recommendation X.25, Interface between data terminal equipment (DTE) and data circuit equipment (DCE) for terminals operating in the packet mode and connected to public data networks by dedicated circuit.* CCITT, Geneva, (1984)

[Chesson88] Chesson, G. "Packet Driver Protocol" *Bell Laboratories Internal Memorandum* updated by J. Gilmore, (March 1988)

[Christensen82] Christensen, W. *Xmodem protocol overview*. **XMODEM.DOC**, (September 1982)

[Cole82] Cole, R. *Computer Communications*. MacMillan, Basingstoke, Hampshire, (1982)

[Collinson86] Collinson, P. "TERM — A Multiplexing Terminal Eumlator for the Atari ST" *Computing Laboratory Internal Report 39*, University of Kent at Canterbury, (July 1986)

[Comer79] Comer, D. "The Ubiquitous B-Tree" *ACM Computer Survey* **11**, 2, (June 1979), pp 121-137

[Compuserve85] CompuServe Inc. *VIDTEX 4.0 Standards for Terminal Emulator Programs*. **BPROTO.DOC**, CompuServe Inc., Colombus, Ohio, (April 1985)

[Compuserve87] CompuServe Inc. *The CompuServe Quick B Protocol*. CompuServe Inc., Colombus, Ohio, (November 1987)

[Cunningham85] Cunningham, M. *File Structure and Design*, Chartwell-Bratt, Bromley, Kent, (1985)

[da Cruz84] da Cruz, F. and Catchings, W. "KERMIT — a simple File Transfer for Microcomputers" *Byte* **9**, 6/7, (June/July 1984), pp 255-278/143-145,400-403

[da Cruz87] da Cruz, F. *Kermit, a file transfer protocol*. Digital Press, Bedford, Massachusetts, (1987)

[da Cruz88] da Cruz, F. "Re: Kermit history (CRC checksum)" *Private communication*, (May 1988)

[DEC86] Digital Equipment Corporation *Guide to VAX/VMS File Applications*, AI-Y508B-TE, (1986)

[Dellar82] Dellar, C. "A File Server for Microcomputers" *Software — Practice and Experience* **12**, 11, (November 1982), pp 1051-1068

[Dion82] Dion, J. and Mitchell, J. "A comparison of two network-based file servers" *Communications of the ACM* **25**, 4, (April 1982), pp 233-245

[FEL88] FEL Computing, *Möbius — Micro to Host Integration*. Undated promotional material from Intelink Data Ltd., Emsworth, Hants., obtained March 1988.

[Forsberg87] Forsberg, C. *Xmodem/Ymodem Protocol Reference*. **YMODEM5.DOC**, Omen Technology Inc., Portland, Oregon, (March 1987)

[Forsberg87b] Forsberg, C. *The Zmodem Inter-Application File Transfer Protocol*. **ZMODEM.DOC**, Omen Technology Inc., Portland, Oregon, (March 1987)

[Frazer79] Frazer, C. ''A compact, portable, CRT-based editor'' *Software — Practice and Experience* **9**, 5, (February 1979), pp 121-125

[Haugdahl84] Haugdahl, J. ''Local-area networks for the IBM PC'' *Byte* **9**, 13, (December 1984), pp 147-178

[Hodges88] Hodges, P. ''The DATAMATION Connectivity Survey: The Haves and Have-Nots'' *Datamation* **34**, 5, (1st March 1988), pp 70-76

[Humphrey88] Humphrey, J. and Smock, G. ''High-Speed Modems'' *Byte* **13**, 6, (June 1988), pp 102-113

[Hunter82] Hunter, J. and Hall, N. ''A Network Screen Editor Implementation'' *Software — Practice and Experience* **12**, 9, (September 1982), pp 843-856

[IBM83] IBM Corporation *MVS/370 Data Management Services*. GC26-4058-0, (1983)

[ISO84] International Standards Organisation *FTAM: File Transfer and Access Method*, DIS 8751/1-4, (1984)

[Jennings86] Jennings, F. *Practical Data Communications: modems, networks and protocols*, Blackwell Scientific Publications, Oxford, (1986)

[Jesty85] Jesty, P. *Networking with Microcomputers*. Blackwell Scientific Publications, Oxford, (1985)

[Kernighan76] Kernighan, B. and Plauger, P. *Software Tools*. Addison-Wesley, Reading, Massachusetts, (1976)

[Kernighan78] Kernighan, B. and Ritchie, D. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, (1978)

[Kernighan78] Kernighan, B. *Advanced Editing on UNIX*, Bell Laboratories Internal Memorandum, (August 1978). Reproduced in *Unix Programmers Manual — Supplementary Documents (4.2 Berkeley Distribution)*, University of California, Berkeley, California, (1984)

[Lapidus72] Lapidus, G. "MOS/LSI launches the low-cost processor" *IEEE Spectrum* **9**, 11, (November 1972), pp 33-40

[Lawrence87] Lawrence, D. and Straight, B. *MicroEMACS Full Screen Text Editor Reference Manual*, (1987)

[Libes78] Libes, S. "The First Ten Years of Amateur Computing" *Byte* **3**, 7, (July 1978), pp 64-71

[Madron87] Madron, T. *Micro-Mainframe Connection.* Howard W. Sams, Indianapolis, (1987)

[Maller79] Maller V. "The Contents Addressable File Store — CAFS" *ICL Technical Journal* **1**, 3, (November 1979), pp 265-279

[Martin76] Martin, J. *Telecommunications and the Computer, 2nd Edition.* Prentice-Hall, Englewood Cliffs, New Jersey, (1976)

[McManis85] McManis, C. "Local power in a remote link" *Byte* **10**, 13, (December 1985), pp 251-258

[McNamara77] McNamara, J. *Technical Aspects of Data Communication.* Digital Press, Bedford, Massachusetts, (1977)

[Meiners87] Meiners, P. *MEGAlink — A File Transfer Protocol.* **MEGALINK.DOC**, P & M Software Company, Huston, Texas, (April 1987)

[Mier84] Mier, E. "The Evolution of the Standard Ethernet" *Byte* **9**, 13, (December 1984), pp 131-142

[Milligan71] Milligan, Spike *A Book Of Milliganimals.* Puffin Books, Penguin Books Ltd., Harmonsworth, Middlesex, (1971), pp 28-29

[Nixon86] Nixon, T. *Fast — a file transfer protocol for use over error-free communications links*. **FAST.DOC**, Hayes Microcomputer Products Inc., Norcross, Georgia, (July 1986)

[NCR86] NCR Corporation *ITX Operating System Reference Manual Release 4*. DI-0545-A, (1986)

[Nowitz82] Nowitz, D. and Lesk, M. "Implementation of a Unix network" *Computer Communications* **5**, 1, (February 1982), pp 30-34

[Ogdin74] Ogdin, J. "Microcomputers: Promises and Practices" *1974 IEEE Intercon Technical Papers*, (March 1974), 17/1 pp 1-12

[Parker87] Parker, J., Kennard, A., and King, D. "The 'Window' terminal" *Computer Journal* **30**, 6, (December 1987), pp 558-564

[Pechara81] Pechara, M. "Microcomputers as Remote Nodes of a Distributed System" *Communications of the ACM* **24**, 11, (November 1981), pp 734-738

[Pike84] Pike, R. "The Blit Terminal" *AT&T Bell Laboratories Techinical Journal* **63**, 8, (October 1984), pp 1607-1631

[Pike87] Pike, R. "The Text Editor sam" *Software — Practice and Experience* **17**, 11, (November 1987), pp 813-845

[Quarterman86] Quarterman, J. and Hoskins, J. "Notable Computer Networks" *Communications of the ACM* **29**, 10, (October 1986), pp 932-971

[Ramsdell80] Ramsdell, R. "The Power of VisiCalc" *Byte* **5**, 11, (November 1980)

[Ritchie78] Ritchie, D. "UNIX Time-Sharing System: A Retrospective" *Bell System Technical Journal* **57**, 6, (July/August 1978), pp. 1947-1969

[Roussel87] Roussel, C. "Re: Inquiry on the program 'blast'", USENET newsgroup *comp.protocols.misc*, <2542@dcatla.UCP>, (December 1987)

[Sedgewick83] Sedgewick, R. *Algorithms*. Addison-Wesley, Reading, Massachusetts, (1983)

[Stallman81] Stallman, R. ''EMACS — The Extensible, Customizable, Self-Documenting Display Editor'' *ACM SIGPLAN Notices*, **16**, 6, (June 1981), pp 147-156

[Stallings85] Stallings, W. *Integrated Services Digital Network.*, IEEE Computer Society Press, Washington, D.C., (1985)

[Sun86] Sun Microsystems Inc., *NFS Protocol Specification.* Sun Microsystems Inc. Mountain View, California, (February 1986)

[Sun86b] Sun Microsystems Inc., *Commands Reference Manual.* Sun Microsystems Inc. Mountain View, California, (February 1986)

[Tanenbaum81] Tanenbaum, A. *Computer Networks.* Prentice-Hall, Englewood Cliffs, New Jersey, (1981)

[Tootill87] Tootill, P. ''Clearing the line'' *Personal Computer World* **10**, 4, (April 1987), pp 174-176

[Tymshare83] Tymshare Inc., *X.PC Protocol Specification.* **X-PC.DOC**, Tymshare Inc. Network Technology Division, Cupertino, California, (September 1983)

[Welch84] Welch, T. ''A Technique for High-Performance Data Compression'' *IEEE Computer* **17**, 6, (June 1984), pp 8-19

[Williams82] Williams, G. ''A Closer Look at the IBM Personal Computer'' *Byte* **7**, 1, (January 1982), pp 36-68

[X/Open85] X/Open *X/Open Portability Guide*, Elsevier Science Publishers B.V., Amsterdam, (1985)

[Zimmerman80] Zimmerman, H. ''OSI Reference Model — The ISO Model of Architecture for Open Systems Interconnection'' *IEEE Transactions on Communication* **COM-28**, 4, (April 1980), pp 425-432

[Zortech87] Zortech Ltd. *Zortech B-Tree ISAM Database Library*, Zortech Ltd, London, (1987)

Appendix A

A serial port interface and terminal emulator

1. Introduction

The operating software on the IBM Personal Computer family and compatibles, MS-DOS, relies on the presence of a selection of routines performing basic functions. These routines, the **B**asic **I**nput **O**utput **S**ystem or BIOS, are stored on in ROM in the PC and perform such functions as reading the keyboard, writing to the screen and reading and writing disc sectors. The BIOS provides routines to monitor the serial ports, returning such information as port status and any character available for reading.

However, these BIOS routines merely provide a snapshot of the state of the port. They do not buffer incoming or outgoing characters, and it is easy without very careful programming to miss incoming characters or fail to maintain maximum output rates at higher line speeds. The underlying hardware allows the port to interrupt the central processor whenever it receives a character or is free to send another. These interrupts can be used by suitable software to provide an improved, buffered interface.

This appendix describes SERINT, a resident serial port handler that does just that, providing a reliable, buffered interface to one of two serial ports. The port may then be run at line speeds of up to 19200 bps with minimal processor overhead and without dropping characters. Also described is a terminal emulator program, VT100, which uses SERINT to provide, as its name suggests, a partial emulation of a DEC VT100 terminal.

1.1. SERINT Installation

SERINT is a terminate and stay resident program (TSR). This category of program, when run under MS-DOS, does not release all the memory assigned to it while it was running, but rather retains some memory permanantly. In the case of SERINT this retained memory is used to keep an input and output buffer and the code necessary to respond to interrupts from the serial port and interface to user software. When SERINT is run, it first checks to see if it has already been installed. If not, it initialises the serial port and exits, staying resident as described, with the message "Serial driver installed". Otherwise it exits

without doing anything with the message ''Serial driver already installed''.

1.2. The User Software Interface

Once SERINT has been installed, it communicates with other software requiring its services through software interrupt 61 (hexadecimal). Six different functions are available; receive a single character, send a single character, set and return port parameters, receive a buffer of characters, send a buffer of characters and return SERINT version number. Parameters and results are passed in registers. This use of registers and software interrupts is employed by MS-DOS itself, and so is supported by virtually all compilers available for the PC.

Details of the available functions and parameters follow. It is assumed the reader is familiar with the 8086 and its registers. Note the use of the carry flag C in returning an error status — it is set on failure and cleared on success.

Function — read character from port

Input AH = 0.

Output AL = character, C = 0. If no character available, C = 1.

Function — write character to port

Input AH = 1, AL = character.

Output C = 0. If buffer was full C = 1 and character not sent.

Function — set and return port settings and status

Input AH = 2, required port settings as per Table A.1.

Illegal values leave current setting unchanged.

Output New port settings and status as per Tables A.1 and A.2.

Register	Controls	Values (*Settings*)
AL	Line speed	0(*300*) 1(*1200*) 2(*2400*)
		3(*4800*) 4(*9600*) 5(*19200*)
BH	Parity	0(*None*) 1(*Odd*) 2(*Even*)
BL	Stop bits	0(*1*) 1(*2*)
CH	Word length	0(*5*) 1(*6*) 2(*7*) 3(*8*)
CL	Port	0(*COM1*) 1(*COM2*)
DH	XON/XOFF	0(*Disabled*) 1(*Enabled*)

Table A.1: Register values and port settings

Register	Returns	Values (*Settings*)
DL	Flow control status	0(*Nothing stopped*) 1(*Send stopped*)
		2(*Receive stopped*) 3(*Both stopped*)
SI	No. characters in input buffer	
DI	No. characters in output buffer	

Table A.2: Returned port status

Function — receive into buffer

Input AH = 3, CX = Buffer size, DS:DX = buffer start

Output CX = No. characters placed in buffer

Function — send from buffer

Input AH = 4, CX = Buffer size, DS:DX = buffer start

Output CX = No. characters taken from buffer

Function — return SERINT version number

Input AH = 128

Output AH = Major version no., AL = Minor version no.

1.2.1. Reading current port settings and resetting port

Calling SERINT function 2, set and return port settings, with illegal values for all port settings leaves everything unchanged and returns the current port status and settings. The recommended procedure for reading the port settings without changing them is to call function 2 with registers BX, CX, DX and AL all set to -1.

Specifying any legal port value will reset the port, even if the value is that of the port currently in use. Resetting the port clears the input and output buffers and resets flow control status to *Nothing stopped.*

1.2.2. Flow control

Table A.1 hints that SERINT is capable of performing XON/XOFF flow control automatically if requested, and this is indeed the case. If XON/XOFF flow control is enabled, then whenever SERINT receives an XOFF character it suspends transmission until an XON character if received. Similarly, if its input buffer fills beyond a certain point, the high water mark, it sends an XOFF character. When the number of characters in the input buffer falls below another point, the low water mark, an XON character is sent. XON and XOFF are ASCII DC3 (Control-S) and DC1 (Control-Q) respectively.

No other for of flow control is supported. This includes hardware methods such as RTS/CTS.

1.2.3. Initial SERINT settings

The initial values SERINT assumes in installtion are as follows.

Setting	Initial value
Line speed	9600
Parity	None
Stop bits	1
Word length	8
Port	COM1
Flow control	Enabled

Table A.3: SERINT intial values

2. VT100

VT100 is a terminal program for use with SERINT. It recognises a subset of the escape codes used by the popular DEC VT100 series of terminals; some of these codes are also ANSI standard screen control codes. It requires the SERINT interface to be present, and allows adjustment of serial port parameters (line speed, word length etc). The set of escape sequences recognised were chosen to allow VT100 to be used effectively with the Unix screen editor *vi* and also the VMS editor *edt*.

VT100 can be instructed to recognise and print *MMMS* packets, and also to make up such packets and send them.

2.1. Limitations

VT100 only supports a subset of the DEC VT100 series escape codes. In some cases this is due to the IBM PC Colour Graphics Adaptor (the only video adaptor for the PC supported by VT100) not physically being able to display the required screen image (no 132 column mode or double height or double width characters, for example), but in the main because VT100 is intended only as a usable terminal program for the local (UKC) environment, and the effort involved in producing a complete DEC VT100 emulation (so far as is possible on the PC) would be considerable.

2.2. Use

On running, VT100 first checks for the presence of the SERINT serial port interface. If this is not found, an error message is printed and VT100 exits. Otherwise the screen is cleared and the bottom line set up as a status line (it is otherwise unused).

From then on, the serial port is continually monitored for incoming characters, which are displayed on the screen subject to any ongoing escape sequences. The keyboard is also monitored and any keystrokes are sent to the serial port, subject to the provisions below.

2.3. Display attributes

The DEC VT100 series has many display capabilities that cannot be reproduced on the PC colour card. The capabilities supported are:

Capability	Shown as
None	Yellow text
Bold	White text
Reverse video	Reverse video
Underline	Reverse video
Blink	Blink

Table A.4: Display capability mapping

2.4. The keyboard

The standard IBM PC keyboard and the standard DEC VT100 keyboards differ in several ways. For example, the PC keyboard has 10 function keys, whereas the DEC VT100 has 4. Also, the DEC keyboard has separate arrow keys, whereas on the PC these are combined with the numeric keypad.

The numeric keypad on the PC keyboard either generates cursor movement and screen control codes (e.g. PgUp, Home etc.) or, if the Num Lock key is toggled, digits. The DEC keyboard has

separate cursor movement keys, and in addition its keypad can generate either digits or 'application' codes — the latter are requested by applications such as *edt* that are controlled using the keypad. There is also the question of the four DEC function keys, PF1 .. PF4.

The emulator adopts the key mappings listed below. Note in particular that the DEC keypad codes are 'application' codes, and must be generated using the number keys on the main keyboard. This is because unfortunately MS-DOS does not return key codes on keystrokes at the keypad that reflect whether the Alt key is depressed. It is possible to extract this information, but the programming effort has yet to be made as far as the emulator is concerned. This also accounts for some of the rather unsatisfactory mappings chosen.

DEC VT100 key	Corresponding PC key(s)
PF1 .. PF4	*f*1 .. *f*4
Keypad 0 .. 9	Alt-0 .. 9†
Keypad - (minus)	*f*5
Keypad , (comma)	*f*6
ENTER	*f*7
Keypad . (period)	*f*8

Table A.5: Keypad mappings

PC keys performing other functions are listed in Table A.6 overleaf.

† The number keys on the main keypad only — this won't work using the keys on the numeric keypad, even with NUMLOCK on.

Key	Function
*f*9	Go to SETUP screen
*f*10	Exit VT100
INS	Toggle status line
Alt-*f*1	Toggle MMMS mode
Alt-*f*2	Send an MMMS packet
Alt-*f*3	Send MMMS ACK packet

Table A.6: Other special keys

2.5. The SETUP screen

This screen allows several terminal parameters ranging from line speed to tab positions to be altered. The current field for adjustment is displayed highlighted, and the current setting may be altered with the up or down arrow keys which will cycle through the available values. When setting tab stops, the left and right arrow keys to move along a screen line, and tab stops may be set or unset using the up or down arrow keys. The TAB key is used to move to the next field (shift TAB moves to the previous field), and *f*9 or *f*10 exit SETUP back to the main screen (the contents of which are not preserved during SETUP).

The following options and settings may be altered during SETUP.

— *Port settings*. That is, line speed, word length, parity, stop bits, which serial port to use and whether automatic XON/XOFF flow control is enabled. VT100 imposes no default port settings.

— *Status line*. This may be toggled off or on. This may also be done using the INS key. Default is ON.

— *Local echo*. Enables/disables echoing of characters typed at the keyboard to the screen. Default is OFF.

— *Auto wrap*. If enabled, an overflowing screen line is continued on the next line. If disabled, the overflowing characters are lost. Default is ON.

— *Logging*. If enabled, all received characters (including escape sequences and control characters) are logged to a file *vt100.log*. The log file is cleared when logging is first enabled in a VT100 session, but logging may be subsequently disabled and re-enabled without losing log file contents. Default is OFF.

2.6. MMMS mode

VT100 can be set to recognise MMMS packets, and to send them under operator control. To do this MMMS mode must be first be engaged with Alt-*f*1, which toggles MMMS mode on or off. VT100 issues a short bleep to reassure the user that MMMS mode has indeed been toggled — the bleep varies in pitch, being high if MMMS mode has been switched on and low if it has been switched off. Note that VT100 assumes the MMMS standard start of packet character (SOH) is in use, and it is not currently possible to instruct VT100 that another control character is being used.

When in MMMS mode whenever the standard MMMS start of packet character is received VT100 displays the information following as an MMMS packet, showing packet type, packet and acknowledgement numbers, packet length and contents, the packet checksum and (if it is different) the calculated checksum. The packet contents are displayed 'as is' — quote characters are not decoded.

MMMS packets may be sent by typing Alt-*f*2. The user is prompted for packet type, packet and acknowledgement numbers and packet contents. VT100 then sends this information as a packet, adding length and checksum information.

Finally, an MMMS acknowledgement packet (packet type 'E') may be sent by hitting Alt-*f*3. This automatically sends a packet acknowledging the last MMMS packet to be received. Note that the last packet is acknowledged; no attempt is made to keep track of the sequence of packets received.

2.7. Escape sequences recognised by VT100

The following escape sequences are recognised by VT100, and acted upon as far as possible. Several escape sequences include numeric values, which are invariably optional (a lack of a parameter leads to some sensible default). Such numeric parameters are specified as 'Pn', and if several are possible this is indicated as 'Pn .. Pn'. Where the meaning of the numeric value depends upon its position some attempt is made to give the meaning in the specification; for example, 'Pl ; Pc' indicates the first parameter is a line number and the second is a column number. Numeric values are always specified in decimal, and multiple values are separated with ';'. For example, the SGR sequence ESC '[0 ; 1 ; 5 m' would switch off all display attributes, and then switch on bold and underline. Escape sequences are always prefaced by the ASCII ESC character.

Those sequences which are ANSI standard sequences are indicated. The sequences are in no particular order. Full details of sequence parameters and their effects are not given.

Name	Sequence	Action	ANSI
HTS	ESC H	Set horizontal tab	•
TBC	ESC [Pn g	Clear horizontal tab	•
EL	ESC [Pn K	Erase in line	•
ED	ESC [Pn J	Erase in display	•
DCH	ESC [Pn P	Delete character(s)	•
IL	ESC [Pn L	Insert line(s)	•
DL	ESC [Pn M	Delete line(s)	•
SGR	ESC [Pn .. Pn m	Set display attributes	•
CUU	ESC [Pn A	Cursor up	•
CUD	ESC [Pn B	Cursor down	•
CUF	ESC [Pn C	Cursor forward	•
CUB	ESC [Pn D	Cursor back	•

Table A.7: Continued ...

Name	Sequence	Action	ANSI
CUP	ESC [Pl ; Pc H	Cursor position	•
IND	ESC D	Index	•
RI	ESC M	Reverse index	•
NEL	ESC E	Next line	•
Set DECCKM	ESC [? 1 h	Send keypad application codes	
Set DECAWM	ESC [? 7 h	Auto-wrap on	
Set IRM	ESC [4 h	Character insert mode on	•
Reset DECCKM	ESC [? 1 l	Send ordinary keypad codes	
Reset DECAWM	ESC [? 7 l	Auto-wrap off	
Reset IRM	ESC [4 l	Character insert mode off	•
DECSC	ESC 7	Save cursor position	
DECRC	ESC 8	Restore cursor position	
DECSTBM	ESC [Pt ; Pb r	Set top and bottom margins	

Table A.7: Recognised escape sequences

All escape sequences not in the above table are ignored, and may or may not be partially printed.

The attribute codes used by the SGR sequence are as follows.

Code	Attribute
0	All off
1	Bold on
4	Underline on
5	Blink on
7	Reverse on

Table A.8: Screen attribute codes

The following ASCII control codes are also acted upon. All others are ignored.

Code	Action
ESC	Receive escape sequence
HT	Tab to next tab stop
BS	Cursor left one space
DEL	Delete char on left then cursor left
BEL	Sound the bell
LF	Line feed
CR	Carriage return
SO	Print graphics characters for 'a' .. 'z'
SI	Print characters 'a' .. 'z' as normal

Table A.9: Recognised control codes

2.8. Use with FlAP starter protocol

VT100 also recognises the initial request for a FlAP application to be started as described in §3.3.3. Ideally, VT100 should complete the starter protocol transactions, obtaining the name of the FlAP application and its parameters, and then start the relevant program.

However, to maximise the amount of memory available to application programs, a crude overlay system has been adopted. A small loader program, TERMINAL, is used rather than invoking VT100 directly. When run, TERMINAL loads and runs VT100. If VT100 encounters the initial FlAP starter request, it exits, passing a special exit status back to TERMINAL. TERMINAL then completes the protocol transactions, executes the FlAP application and re-runs VT100 again.

This method of working is not perfect; in particular, exiting and restarting VT100 restores default terminal settings. Its merit is simplicity and implementation convenience.

Appendix B

URAT Programmer's Manual

This page intentionally left blank

NAME

Intro – introduction to URAT library facilities

DESCRIPTION

URAT (Unix Record Access rouTines, pronounced 'you rat') is a library allowing record-based files to be created and accessed under Unix. A variety of record formats and access modes (including indexed files) are provided, together with a consistent interface to the record handling functions.

RECORD FORMATS AND ACCESS

The *filetype* of a URAT file is determined by two factors, the record format and the desired method of access. Eleven filetypes are available, offering a variety of access methods to five different record formats. *Filetypes*(URAT) lists the available filetypes.

RECORD CURRENCY

URAT tries to access records in a file in a manner similar to the way Unix accesses individual bytes in a file. Records are numbered from 0 (i.e. record 0 is the first record in the file), and URAT maintains a *currency*; the current record being the one which will be read next if no action is taken to change the currency. The notion of currency also avoids having to specify a record number with each record operation. In the case of indexed files, the currency is the position of the current record in the current index.

In some circumstances, chiefly after some indexed file operations, it is impossible to know the current record number. In these cases, the record number is given the value DONTKNOW.

If an error occurs during a record operation, currency may be lost (see *Errors*). In this case, currency should be re-established by using *urSeek* to restore currency to a known position e.g. the start of the file.

LIST OF FUNCTIONS

Record operations are carried out on file of all types using one set of record operations. Some operations may not be available for a given filetype, and other operations may be restricted in some manner; for example, FINDing a record is only applicable to indexed files, and whilst it is possible to SEEK directly to the end of a variable-length record sequential file, it is not possible to know how many records have been passed over.

Available functions are:

Name	*Description*
urAddRec	Add a new record
urClose	Close a record file
urCreate	Create and open a new record file
urDeleteFile	Delete a record file
urDeleteRec	Delete a record
urFindRec	Locate a record by searching an index
urGetCurrency	Save the currency
urGetRec	Read the current record
urOpen	Open a record file
urRenameFile	Rename a record file
urSeek	Move to a record
urSetCurrency	Reset currency to saved value
urTruncate	Truncate a record file
urUpdateRec	Update a record

Other manual pages:

Errors	URAT error codes
FileTypes	Available URAT file types
Intro	This page
RecordSpec	Specifying record and index details

BUGS AND PROBLEMS

The library is currently organised as a module for each file type and an interface module which performs some parameter checking and dispatches control to the correct handling routine for the file type in use. As all uses of the library are routed through the same functions, it is necessary to include the whole library in any application using it. The library is not small (currently around 70k); it seems wasteful for a simple

sequential file processing program to be carrying around the baggage of code for handling indexed files.

Indexed files currently use a separate Unix file for each index (variable-length indexed files have an extra index handling space allocation in the data file). urDeleteFile and urRenameFile will delete and rename respectively the data file and all indexes.

The file structure used for relative text files keeps an index of where in the main data file records start. While the file is open this index is held in memory — should anything go wrong and the file not be properly closed, the index will be lost and the file corrupted beyond recovery. This file type should really only be used for temporary files.

This is an initial version of URAT, and currently is undoubtedly stuffed with bugs. These may or may not get fixed — URAT is really just a research tool, and will certainly be forgotten forever once its usefulness is at an end. It is also rather slow.

ACKNOWLEDGEMENTS

URAT uses the Zortech BTree and Vlen libraries.

NAME

urErrno – URAT error codes

SYNOPSIS

#include "urat.h"

extern int urErrno ;

DESCRIPTION

On failure, all URAT functions return an indication that an error has occurred. Further information on the error may be found in *urErrno*, which is set on return from each URAT function.

CURRENCY

In general, if a URAT function fails the file currency is not defined. In other cases the file currency may vary depending upon the file type. For example, using *urSeek* to seek to record 10 in a file which only has 9 records will fail with error EEOF if the file type does not permit seeking beyond the end of file. For an indexed file, currency is unchanged; a variable-length record sequential file has to be scanned through to find the record number of the last record in the file, and so currency on failure is at EOF.

ERROR CODES

urErrno may take the following values:

[EALLOK] The last URAT function completed without error.

[EBADCUR] The currency passed to *urSetCurrency* is not valid.

[EBADFD] The given file descriptor does not correspond to an open file.

[EBADKEY] The key specification given for this index file is malformed.

[EBADMATCH] Unknown match type given to *urFindRec*.

[EBADNAM] The given file name is not a proper file name.

[EBADOP] Invalid operation for a file of this type.

[EBADORG] The structure of the named file does not correspond to that given (e.g. attempting to

open sequential file as an indexed file).

[EBADPTR] A pointer (to a filename of data buffer) points to outside the process data space.

[EBADRLEN] Record length is too short or too long.

[EBADSIZ] A read or write of a negative number of bytes has been requested.

[EBADSK] Invalid whence parameter to *urSeek*.

[EBOF] At the start of the file.

[ECNTBKUP] Can't seek backwards from the end of a variable-length record file.

[ECNTDEL] Couldn't delete file.

[ECNTREW] Rewind of a sequential file failed.

[ECNTTRNC] Attempt to truncate file failed.

[ECNTXTND] The file cannot be extended, usually because no space is available on the file system.

[EEOF] Either an attempt is being made to read beyond the end of a file, or to seek beyond the end of the file when this is not permitted for the file type.

[EFLNTFND] No file with that name can be found.

[EIDXSPC] Failure allocating index space for new entry to file.

[EINTRNL] Ghastly internal error in URAT. Usually a problem allocating memory. If this error should occur, your file *may* have been corrupted.

[ENODUPS] Duplicate keys not allowed in this indexed file. The index and file are reset to their state before the record operation started.

[ENOIDX] No such index.

[ENOKEY] No primary key specified for this indexed file.

[ENOORG] No such file organisation is supported.

[ERECBAD] Error reading a record (usually because end of file occurred before the end of the

record).

[ETOMNYFLS] Too many files open.

[ETOMNYKEYS] Too many keys defined for index file.

[EUNIXERR] Miscellaneous Unix error not covered by the above.

NOTES

The Unix error code in *errno* may be inspected for further clues to why a failure occurred. However this may not be useful if URAT has taken some corrective action after the error, as is sometimes the case.

urErrno is **always** set by a call to a URAT function.

NAME

Filetypes – available URAT file types

SYNOPSIS

#include "urat.h"

DESCRIPTION

Currently 11 file types are defined for URAT. They are classified into 4 record types as follows:

Fixed-length records are records which occupy a fixed amount of space per record. Sequential (*FL_SEQ*), Relative (*FL_REL*), and Indexed (*FL_IND*) files are available. There is no distinction between FL_SEQ and FL_REL files in the current Unix implementation.

Variable-length records are records which occupy a varying amount of space per record. The normal variable-length record format, format 1, has the following file types available: Sequential (*VL1_SEQ*), Relative (*VL1_REL*), and Indexed (*VL1_IND*). An alternative variable-length record format, format 2, is available for sequential files only (*VL2_SEQ*). This format complies with American National Standard X3.27-1977, and should be used when creating conforming files for tape transfer.

Text records are variable-length records whose sequential format corresponds to that for text files under Unix. Two file types are available, Sequential (*TXT_SEQ*), and Relative (*TXT_REL*). Relative text files offer *urAddRec*(URAT) and *urDeleteRec*(URAT) facilities more appropriate to text handling, at some expense in disc space and processing time. Their use should be restricted to temporary files into which sequential text files are read for processing and the text copied back to a sequential file on termination. Currently, they are wasteful of disc space (extending a record allocates fresh data space for the record, and the unused space is never reclaimed) and very fragile (see *Intro*(URAT)).

Bytestream files have no record structure at all. Multiple bytes may be read or written in one operation, and *urSeek*(URAT) deals in byte offsets. Sequential (*STREAM_SEQ*) and Relative (*STREAM_REL*) file types are available, though there is no distinction between the two under the current Unix implementation.

NOTES

The normal variable-record format, format 1, is to preface the record itself with a two-byte binary count of

the bytes in the record itself (not including the count preface), the count being stored low-byte – high-byte. This count is invisible to the record-processing operations. Note that relative variable-length files must have a maximum record length specified when the file is opened, and records are stored in fixed-size cells big enough to hold a maximum-length record. The maximum record length specified should not include space for the record count prefix; URAT handles all such matters internally.

Variable-length record format 2 prefaces each record with a 4–byte decimal count of the record length, inclusive of the count. URAT record processing operations adjust the record length to that of just the record data.

SEE ALSO

Recordspec(URAT)

NAME

Recordspec – record details specification

SYNOPSIS

```
#include "urat.h"

struct KeySeg
{
        u_short start ;
        u_short length ;
}

struct KeySpec
{
        u_short flags ;
        u_short no_segs ;
        struct KeySeg    seg [MAXSEGS] ;
}

struct RecSpec
{
        u_short reclen ;
        u_short no_keys ;
        struct KeySpec   *kspec ;
}
```

DESCRIPTION

The record length of a fixed-length record file is given in *reclen*. For a variable-length record file *reclen* specifies the maximum record length permissible. This is mandatory for a relative file, but optional for sequential and indexed files and may be set to 0 to indicate no maximum record length for VL1_SEQ, VL2_SEQ, TXT_SEQ and TXT_REL files. A non-zero record length specified for a bytestream file

(STREAM_SEQ, STREAM_REL) just imposes an upper limit on the number of bytes that may be read during one *urGetRec*(URAT) operation.

Indexed files are indexed on one *primary key*. They may also be indexed on further alternate keys, up to a maximum of *MAXKEYS* keys (including the *primary key*). *no_keys* gives the number of keys on which to index, and must be a minimum of 1 (the *primary key*).

A *key* is made up from up to *MAXKEYSIZE* printable ASCII characters (i.e. it may not contain control characters). The format of each key is specified by the corresponding entry in an array of key specifications pointed to by *kspec* (so the primary key specification is at kspec[0], the next key specification at kspec[1] etc.). Up to *MAXKEYS* keys are supported.

A key is made up from 1 to *MAXSEGS* segments, a segment being a number of bytes from the record starting at a given offset. Hence a key may be made up from several different parts of the record. For fixed-length records and variable-length records for which a maximum record length is given, each segment must lie wholly inside the record. Should a variable-length record not be long enough to wholly or partially contain the segment, the key value for that segment will be padded with spaces. Similarly, if the record contains non-printable characters these are replaced in the key with spaces.

flags should be set to either *DUPS_OK* or *NO_DUPS* to indicate whether duplicate keys should be permitted, or whether any attempt to add a record with a duplicate key should be rejected. Forbidding duplicate keys saves some disc space for the index; allowing duplicate keys restricts maximum key size to *MAXKEYSIZE* - 3 characters, as 3 characters are automatically added on to the key to ensure duplicate keys are kept separate).

NOTES

The current value of *MAXKEYS* is 8, the current value of *MAXSEGS* is also 8, and the current value of *MAXKEYSIZE* is 80.

SEE ALSO

Filetypes(URAT)

NAME

urAddRec – add a record

SYNOPSIS

#include "urat.h"

int urAddRec(fd, bufsiz, bufptr)

int fd, bufsiz ;

char *bufptr ;

DESCRIPTION

urAddRec adds a new record to the file with descriptor *fd* using up to *bufsiz* bytes from *bufptr*.

If the file is of fixed-length records the record will be either truncated or padded with ASCII NULs if *bufsiz* does not correspond to the record length.

If the file is of variable-length records the record will be truncated if it exceeds the maximum record length specified.

In most cases, *urAddRec* merely locates the current end of file and adds a record.

In the case of indexed files, the new record is added to the data file and all indexes.

For relative text (TXT_REL) files, the new record is inserted into the file *before* the current record. All succeeding records 'move down'; if a record is inserted before record *n*, record *n* becomes record *n*+1.

If the file is a bytestream, *bufsiz* bytes are written, starting at EOF.

CURRENCY

The current record becomes the record after the added record; this will be EOF for all but TXT_REL and indexed files. For indexed files, currency is the record following the added record in the current index.

NOTES

In the case of indexed files, all indexes are updated to reflect the addition of the data record. No provision is made for delaying updating of indexes not currently in use.

RETURN VALUE

If an error occurs, –1 is returned. Otherwise the number of bytes written to the new record is returned.

SEE ALSO

urDeleteRec(URAT), *urUpdateRec*(URAT)

ERRORS

urAddRec will fail and the record will not be added for any of the following reasons:

[EBADFD] The given file descriptor does not correspond to an open file.

[EBADPTR] *bufptr* is not a valid pointer.

[EBADSIZ] *bufsiz* is negative.

[ECNTXTND] Cannot extend file to make room for new record.

[EIDXSPC] No space to add index entry.

[EINTRNL] Ghastly internal error in URAT. Usually a problem allocating memory.

[ENODUPS] Attempting to duplicate a key in an indexed file which does not permit duplicate keys.

[ERECBAD] Bad record found whilst looking for end of file.

[EUNIXERR] Miscellaneous Unix error.

NAME

urClose – close an open record file

SYNOPSIS

#include "urat.h"

int urClose(fd)

int fd ;

DESCRIPTION

urClose attempts to close the record file opened with *fd*. Any file buffers are written to disc, and *fd* becomes inactive. *urOpen* must be called before the file can be used again.

If the file was specified as a temporary file, it is automatically deleted on closing.

CURRENCY

Not applicable; currency lost.

RETURN VALUE

The value –1 is returned if an error occurs. Otherwise 0 is returned.

ERRORS

urClose will fail for any of the following reasons:

[EBADFD] The given file descriptor does not correspond to an open file.

[ECNTXTND] No room on disc to save index buffers.

[EINTRNL] Ghastly internal error in URAT. Usually a problem allocating memory.

NAME

urCreate – create a record file

SYNOPSIS

#include "urat.h"

int urCreate (fname, ftype, spec)

char *name ;

struct RecSpec *spec ;

int ftype ;

DESCRIPTION

urCreate attempts to create a record file of the given file type, and specification with the name at *fname*. File(s) will be created to hold indexes as appropriate.

If a file of that name already exists it is emptied.

A temporary file, placed in /tmp and ensured a unique file name using *fname* as a base name can be created by adding the modifier TEMPFILE to *ftype*. Temporary files are deleted on being closed.

If a non-zero record length is supplied on creating a bytestream file (STREAM_SEQ, STREAM_REL) then that record length is used as an upper bound on the number of bytes read by *urGetRec*(URAT).

CURRENCY

At BOF.

NOTES

Valid file types for *ftype* are listed in *Filetypes*(URAT).

Details of the required record specifications in *spec* are given in *Recordspec*(URAT).

RETURN VALUE

The value –1 is returned if an error occurs. Otherwise a non-negative file descriptor is returned.

ERRORS

urCreate will fail and the file will not be created or truncated for any of the following reasons:

[EBADKEY] Indexed file key specification is invalid — error in number of segments, segment specification or overall key length.

[EBADNAM] *fname* does not point to a valid filename.

[EBADPTR] *fname* is not a valid pointer.

[EBADRLEN] Bad record length given.

[ECNTTRNC] Existing file could not be truncated. This may be because access to the file is not permitted, or because it is a directory.

[ECNTXTND] Ran out of disc space trying to write file header.

[EIDXSPC] No spare memory available for index buffers.

[EINTRNL] Ghastly internal error in URAT. Usually a problem allocating memory.

[ENOKEY] No primary key specified for indexed file.

[ENOORG] No such file organisation is supported.

[ETOMNYFLS] You are trying to open too many files at once.

[ETOMNYKEYS] You can't have more than MAXKEYS indexes for an indexed file.

[EUNIXERR] Miscellaneous Unix error not covered by the above.

NAME

urDeleteFile – delete a record file

SYNOPSIS

#include "urat.h"

int urDeleteFile(fname)

char *fname ;

DESCRIPTION

urDeleteFile deletes the file with the file name given in *fname*.

If the named file is an indexed file, all the file's indexes are deleted too.

NOTES

Errors are not reported if any index file cannot be found. After all, it was going to be deleted anyway.

RETURN VALUE

If an error occurs, –1 is returned. Otherwise 0 is returned.

SEE ALSO

urRenameFile(URAT)

ERRORS

urDeleteFile will fail and the file will not be deleted (assuming it exists in the first place) for any of the following reasons:

[EBADNAM] The give file name is not a properly-constructed filename.

[ECNTDEL] Couldn't delete file — perhaps access permissions not relaxed enough.

[EFLNTFND] No file with that name could be found.

[EINTRNL] Nasty internal error — in this case, couldn't delete index file.

[EUNIXERR] Miscellaneous Unix error.

NAME

urDeleteRec – delete a record

APPLICABLE-FILE-TYPES

FL_IND, VL1_IND, TXT_REL

SYNOPSIS

#include "urat.h"

int urDeleteRec(fd)

int fd ;

DESCRIPTION

urDeleteRec deletes the current record in the file with descriptor *fd*.

CURRENCY

The current record becomes the record after the deleted record.

NOTES

In the case of indexed files, all indexes are updated to reflect the deletion of the data record.

In all but one case, the space made available by the deletion can be reused. However, no garbage collection or compaction occurs. In TXT_REL files, the index space may be reused, but not data space.

After a *urDeleteRec*, all records following the deleted record have their record numbers as used by *urSeek* effectively decremented by one; if record *n* is deleted, records *n*+1 and *n*+2 become records *n* and *n*+1 respectively.

RETURN VALUE

If an error occurs, –1 is returned. Otherwise 0 is returned.

SEE ALSO

urAddRec(URAT)

ERRORS

urDeleteRec will fail and the record will not be deleted for any of the following reasons:

[EBADFD] The given file descriptor does not correspond to an open file.

[EBADOP] Records cannot be deleted in this type of file i.e. file is not of type FL_IND, VL1_IND or TXT_REL.

[EEOF] At the end of the file.

[EINTRNL] Ghastly internal error in URAT. Usually a problem allocating memory.

[EUNIXERR] Miscellaneous Unix error.

NAME

urFindRec – find a record in an indexed file

APPLICABLE-FILE-TYPES

FL_IND, VL1_IND

SYNOPSIS

#include "urat.h"

int urFindRec(fd, index_no, match_type, bufsiz, bufptr)

int fd, index_no, match_type, bufsiz ;

char *bufptr ;

DESCRIPTION

urFindRec works in the same manner as *urGetRec*, except that rather than reading the current record, it first searches for a match to a key constructed from the record at *bufptr*, and returns that record in *bufptr* if a suitable one is found.

index_no specifies the index which is to be searched (request index 0 to search the current index). The selected index remains the current index.

Several match criterion are available via *match_type*.

EXACT	Find the first record with a key exactly matching the given key.
PARTIAL	Find the first record with a key greater than or equal to the given key.
GREATER	Find the first record with a key greater than the given key.
NOSEARCH	Don't find a record, just switch index. Do not even return the current record in that index, just 0 if successful.

CURRENCY

If the find succeeds, the current record is the one after the find, unless the NOSEARCH option was requested, in which case the currency is unchanged. The current record number will be DONTKNOW. If the find fails, the current record is unchanged.

NOTES

In indexes which permit duplicate keys, the oldest duplicate is found. The duplicates can then be stepped through in the order in which they were added using *urGetRec*.

RETURN VALUE

urFindRec returns the number of bytes transferred to *bufptr*. Should the search fail, or encounter any error, *urFindRec* will return –1.

SEE ALSO

urGetRec(URAT)

ERRORS

urFindRec will fail and the record will not be located for any of the following reasons:

[EBADFD] The given file descriptor does not correspond to an open file.

[EBADMATCH] Unknown match type specified.

[EBADOP] Find cannot be used for any file not an indexed file, that is of type FL_IND or VL1_IND.

[EBADSIZ] *bufsiz* is negative.

[EEOF] Find failed — no suitable match (i.e. hit end of index without search succeeding. Currency is as before call to *urFindRec*).

[EINTRNL] Ghastly internal error in URAT. Usually a problem allocating memory.

[ENOIDX] No such index.

[EUNIXERR] Miscellaneous Unix error.

NAME

urGetCurrency – save the currency

SYNOPSIS

#include "urat.h"

int urGetCurrency(fd, cur)

int fd ;

union urCurrency *cur ;

DESCRIPTION

urGetCurrency saves enough information about the currency for file *fd* in the space pointed to by *cur* to enable that currency to be restored later by *urSetCurrency*(URAT).

The union *urCurrency* in which that information is saved is defined in *urat.h*.

urGetCurrency and *urSetCurrency* can be used even when the currency is unknown, e.g. at the result of a *urFindRec*.

CURRENCY

Unchanged.

RETURN VALUE

The value –1 is returned if an error occurs, otherwise the value 0 is returned.

ERRORS

urGetCurrency will fail and the record will not be retrieved for any of the following reasons:

[EBADFD] The given file descriptor does not correspond to an open file.

[EINTRNL] Ghastly internal error in URAT. Usually a problem allocating memory.

[EUNIXERR] Miscellaneous Unix error.

NAME

urGetRec – read a record

SYNOPSIS

#include "urat.h"

int urGetRec(fd, bufsiz, bufptr)

int fd, bufsiz ;

char *bufptr ;

DESCRIPTION

urGetRec reads the current record in the file with descriptor *fd* into the destination indicated by *bufptr*. Up to *bufsiz* bytes are written to the destination buffer — if the record is longer the remaining bytes are ignored. The length of the record is returned.

If *bufptr* is NULL or *bufsiz* is zero then no data is transferred, but the record length is still returned.

If a bytestream file (STREAM_SEQ, STREAM_REL) is created/opened with a non-zero record length, that record length imposes an upper bound on the number of bytes that can be read by one *urGetRec* call. The result for a bytestream file is always the number of bytes read.

CURRENCY

The next record becomes the current record.

NOTES

Unexpectedly long variable-length records could cause the destination buffer to overflow, hence the need for *bufsiz*.

RETURN VALUE

The value –1 is returned if an error occurs. Otherwise the number of bytes in the record is returned (this may be greater than the number of bytes placed in the destination buffer if *bufsiz* is exceeded.

ERRORS

urGetRec will fail and the record will not be retrieved for any of the following reasons:

[EBADFD] The given file descriptor does not correspond to an open file.

[EBADPTR] *bufptr* is not a valid pointer.

[EBADSIZ] *bufsiz* is negative.

[EEOF] At the end of the file — no record to read.

[EINTRNL] Ghastly internal error in URAT. Usually a problem allocating memory.

[ERECBAD] Record is corrupt. Usually means end of file was found before the expected end of record.

[EUNIXERR] Miscellaneous Unix error.

NAME

urOpen – open an existing record file

SYNOPSIS

#include "urat.h"

int urOpen(fname, ftype, reclen)

char *fname ;

int ftype, reclen ;

DESCRIPTION

urOpen attempts to open a record file of the given file type using the name given in *fname*. In cases where the record length *reclen* is required and cannot be deduced from the file (FL_SEQ, FL_REL, VL1_REL) it must be supplied — otherwise *reclen* is ignored.

If a record length is specified for a bytestream file (STREAM_SEQ, STREAM_REL), the only effect is for the record length to be used as an upper limit on the number of bytes returned by a call to *urGetRec*(URAT).

CURRENCY

At BOF.

NOTES

Valid file types for *ftype* are listed in *Filetypes*(URAT). After opening, the current record is the first record in the file.

RETURN VALUE

The value –1 is returned if an error occurs. Otherwise a non-negative file descriptor is returned.

ERRORS

urOpen will fail and the file will not be opened for any of the following reasons:

[EBADNAM] *fname* does not point to a valid filename.

[EBADORG] File is not of the organisation specified.

[EBADPTR] *fname* is not a valid pointer.

[EBADRLEN] Given record length is invalid.

[ECNTTRNC] Existing file could not be truncated.

[EFLNTFND] File (either data file or index file, if appropriate) not found.

[EINTRNL] Ghastly internal error in URAT. Usually a problem allocating memory, but for indexed files may be a problem reading the key specifications.

[ENOORG] No such file organisation is supported.

[ETOMNYFLS] You are trying to open too many files at once.

[EUNIXERR] Miscellaneous Unix error not covered by the above.

NAME

urRenameFile – rename a record file

SYNOPSIS

#include "urat.h"

int urRenameFile(from_fname, to_fname)

char *from_fname, *to_fname ;

DESCRIPTION

urRenameFile renames the file with the file name given in *from_fname* to the new name given in *to_fname*.

If the named file is an indexed file, all the file's indexes are renamed too.

NOTES

Any existing file with a name of *to_fname* will be deleted. Watch it.

RETURN VALUE

If an error occurs, –1 is returned. Otherwise 0 is returned.

SEE ALSO

urDeleteFile(URAT)

ERRORS

urRenameFile will fail and the file will not be renamed (assuming it exists in the first place) for any of the following reasons:

[EBADNAM] One of the given file names is not a properly-constructed filename.

[ECNTDEL] Couldn't rename file — perhaps access permissions not relaxed enough.

[EFLNTFND] No file with that name could be found.

[EUNIXERR] Miscellaneous Unix error.

NAME

urSeek – move to a record

SYNOPSIS

#include "urat.h"

long urSeek(fd, offset, whence)

int fd, whence ;

long offset ;

DESCRIPTION

urSeek moves to the record indicated by *offset* and *whence*, and makes that record the current record.

offset is the number of records away from the starting point given by *whence*, and may be positive or negative; *whence* may be one from

0 *offset* is relative to the start of the file.

1 *offset* is relative to the current position.

2 *offset* is relative to the end of the file.

The new current record number is returned. This may be unknowable, in which case DONTKNOW is returned; this occurs in indexed and variable-length sequential files only. For all variable-length sequential file types, DONTKNOW is **only** returned as the result of a seek to the end of the file (as it is not possible to know the number of records skipped without scanning through them). Indexed file currency may be set to DONTKNOW after an *urAddRec* or *urFindRec*.

Record numbers run from 0 upwards.

Bytestream files (STREAM_SEQ, STREAM_REL) are regarded as having a record length of 1 — hence *urSeek* regards *offset* as a byte offset.

CURRENCY

The new record becomes the current record.

NOTES

Limitations on *urSeek* vary depending upon the type of file in use.

Sequential files: Attempts to seek beyond the end of the file will be curtailed at the end of the file. Using *urSeek* with an offset relative to the end of the file on variable-length record files can only be used if that offset is zero, as it is not possible to know how many records there are in the file and thus how far to seek. Similarly, you can seek to the end of a variable-length file, but as the number of records passed over is not known the current record number is set to DONTKNOW. Note that seeking through variable-length records can only be accomplished by reading each and every intervening record, and seeking backwards can only be accomplished by rewinding the file and reading through forwards.

Relative files: *urSeek* cannot be used to extend the file (by seeking beyond the end-of-file) for relative text files (TXT_REL).

Indexed files: *urSeek* operates on indexed files by stepping through the current index (possibly first moving to the start or end of the index). The current record number may well be unknowable (e.g. after a *urFindRec*(URAT), or a *urAddRec*(URAT)), in which case *urSeek* returns a value of DONTKNOW. Again, any attempt to *urSeek* beyond the end of the file is curtailed at the end of the file.

RETURN VALUE

The value –1 is returned if an error occurs. Otherwise the new current record number is returned (possibly DONTKNOW for indexed files).

ERRORS

urSeek will fail for any of the following reasons:

[EBADFD] The given file descriptor does not correspond to an open file.

[EBADSK] *whence* is invalid.

[EBOF] Can't seek backwards from the start of the file.

[ECNTBKUP] Can't use negative offsets from the end of a variable-length record sequential file.

[ECNTREW] Couldn't rewind file to its start.

[EEOF] Can't move beyond end of file for this file type.

[EINTRNL] Ghastly internal error in URAT. Usually a problem allocating memory.

[ERECBAD] Corrupt record encountered whilst scanning forward through file. Usually end of file has occurred before the expected end of record.

[EUNIXERR] Miscellaneous Unix error.

NAME

urSetCurrency – restore the currency

SYNOPSIS

#include "urat.h"

int urSetCurrency(fd, cur)

int fd ;

union urCurrency *cur ;

DESCRIPTION

urGetCurrency uses the currency information saved by *urGetCurrency* for the file *fd* in the space pointed to by *cur* to restore that currency.

The union *urCurrency* in which that information is saved is defined in *urat.h*.

urGetCurrency and *urSetCurrency* can be used even when the currency is unknown, e.g. at the result of a *urFindRec*.

CURRENCY

Unchanged.

RETURN VALUE

The value –1 is returned if an error occurs, otherwise the value 0 is returned.

ERRORS

urSetCurrency will fail and the record will not be retrieved for any of the following reasons:

[EBADCUR] The currency given by *cur* is invalid (e.g. attempt to set currency to a deleted record).

[EBADFD] The given file descriptor does not correspond to an open file.

[EINTRNL] Ghastly internal error in URAT. Usually a problem allocating memory.

[EUNIXERR] Miscellaneous Unix error.

NAME

urTrucate – truncate the file from the current record onwards

SYNOPSIS

#include "urat.h"

int urTruncate(fd)

int fd ;

DESCRIPTION

urTruncate deletes the current record onwards in the file with descriptor *fd* and truncates the file.

CURRENCY

The current record becomes EOF. In cases of failure the current record is undisturbed.

NOTES

Note that *urTruncate* is available for indexed files, but is achieved by repeated calls to *urDeleteRec*, and so is likely to be timeconsuming and not to have any effect on the size of the file, as the records deleted do not necessarily come from the end of the file. Similarly, truncating a TXT_REL file merely truncates the index; no record space is made available for reuse.

RETURN VALUE

If an error occurs, –1 is returned. Otherwise 0 is returned.

SEE ALSO

urDeleteRec(URAT)

ERRORS

urTruncate will fail and records will not be deleted for any of the following reasons:

[EBADFD] The given file descriptor does not correspond to an open file.

[ECNTTRNC] Couldn't truncate the data file.

[EEOF] At the end of the file.

[EINTRNL] Ghastly internal error in URAT. Usually a problem allocating memory.

[EUNIXERR] Miscellaneous Unix error.

NAME

urUpdateRec – update a record

SYNOPSIS

#include "urat.h"

int urUpdateRec(fd, bufsiz, bufptr)

int fd, bufsiz ;

char *bufptr ;

DESCRIPTION

urUpdateRec updates the current record in the file with descriptor *fd* using up to *bufsiz* bytes from *bufptr*.

If the file is of fixed-length records the record will be either truncated or padded with ASCII NULs if *bufsiz* does not correspond to the record length.

If the file is of variable-length records the record will be truncated if it exceeds the maximum record length specified.

If the file is a bytestream, *bufsiz* bytes are written, starting at the current offset.

If *urUpdateRec* is called when the current record is EOF, it adds a new record to the file (at EOF *urUpdateRec* is equivalent to *urAddRec*, except that index file currency is kept at EOF).

CURRENCY

The current record becomes the record after the updated record. In the case of indexed files, this is the record following the record to be updated **before** updating starts; the updating process may well change the key, and thus the position of the updated record in the file.

NOTES

It is possible to use *urUpdateRec* on sequential variable-length record files, but the practice is heavily discouraged. Unless the updated record is the same length as the old record, the rest of the file will become hopelessly corrupted.

RETURN VALUE

If an error occurs, –1 is returned. Otherwise the number of bytes written to the file is returned.

SEE ALSO

urSeek(URAT), *urAddRec*(URAT)

ERRORS

urUpdateRec will fail and the record will not be updated for any of the following reasons:

[EBADFD] The given file descriptor does not correspond to an open file.

[EBADPTR] *bufptr* is not a valid pointer.

[EBADSIZ] *bufsiz* is negative.

[ECNTXTND] Can't extend file to make room for new record.

[EIDXSPC] No space in index to add new key entries (on updating, new keys are added before old ones are deleted).

[EINTRNL] Ghastly internal error in URAT. Usually a problem allocating memory.

[ENODUPS] Update is attempting to duplicate a key in an indexed file which does not permit duplicate keys.

[EUNIXERR] Miscellaneous Unix error.

Appendix C

FlAP Programmer's Manual

This page intentionally left blank

NAME

Intro – introduction to FlAP library facilities

DESCRIPTION

FlAP (**F**ile **A**ccess **P**rotocol) is a library of C functions allowing record-based files on a mainframe to be created, manipulated and otherwise accessed from a microcomputer, the two being connected by a terminal line or other slow connection. It comes in two parts; a library which must be linked into microcomputer programs, and a server running on the mainframe that must be started before file operations can proceed.

This manual describes FlAP as embodied in the only current implementation, a Unix file server communicating with IBM PC or compatible microcomputers.

RECORD FORMATS AND ACCESS

A FlAP record file is distinguished by the record format and method of access. Eleven file types are supported. *Filetypes*(FlAP) gives details.

LIST OF FUNCTIONS

Record operations are carried out on files of all types using one set of record operations. Some operations may not be available for a given filetype, and other operations may be restricted in some manner; for example, FINDing a record is only applicable to indexed files, and whilst it is possible to SEEK directly to the end of a variable-length record sequential file, it is not possible to know how many records have been passed over.

Most of the record operations listed below are exactly those provided by *URAT* (see the URAT Programmer's Manual). However, FlAP does provide several higher-level file operations, and extend some other *URAT* functions slightly.

Available functions are:

Name	*Description*
flqAbort	Abort ongoing FlAP operation
flqAddRec	Add a new record

flqVAdd	Add multiple new records
flqClose	Close a record file
flqCompare	Compare successive records in two files
flqCopy	Copy successive records from one file to another
flqCreate	Create and open a new record file
flqDelFile	Delete a record file
flqDelRec	Delete record(s)
flDone	See if FlAP operation has finished yet
flqDuplicate	Duplicate records within a file
flqFindRec	Locate a record by searching an index
flqGetRec	Read the current record
flqVGet	Read multiple records
flInit, flStart, flReady	Establish FlAP communication
flqOpen	Open a record file
flqRenFile	Rename a record file
flqSearch	Search for record with given regular expression
flqSeek	Move to a record
flShutdown	Close down FlAP communication
flqTruncate	Truncate a record file
flqUpdateRec	Update a record
flqVUpdate	Update multiple records
flWait	Wait for a FlAP operation to complete

Other manual pages:

Errors	FlAP error codes
FileTypes	Available FlAP file types
Intro	This page
RecordSpec	Specifying record and index details

Term-Starter	Starting FlAP applications from a Unix command line
Unix-Server	The Unix file server
VectorSpec	Specifying multiple records details

Each flq?????? function has a counterpart fl?????? function that issues the flq????? request and waits for it to complete. The value returned is the result normally obtained from flWait.

BUGS AND PROBLEMS

This is a preliminary implementation, with all that *that* implies. It has not been throughly tested.

Since all parameters must be passed over a slow serial link, life is a tad slow at times. Given the serial link, no solution seems feasible.

It is only easyish to add extra FlAP functions. The parameter passing gets rather hairy at times.

FlAP is supposed to be portable amongst microcomputers and mainframes. These pages describes the only extant implementation, a library for IBM PC series and compatible microcomputers and a file server running under 4.3BSD Unix. Different implementations will undoubtedly reveal implementation-dependent aspects of FlAP which require slight alterations to function interfaces.

NAME

Errors – FlAP error codes

SYNOPSIS

#include "flap.h"

extern int flErrno ;

DESCRIPTION

On failure, all FlAP functions return an indication that an error has occured. Further information on the error may be found in the global variable *flErrno*, which is set on return from each FlAP function.

CURRENCY

In general, if a FlAP function fails the file currency is not defined. In other cases the file currency may vary depending upon the file type. For example, using *flqSeek* to move to record 10 in a file which only has 9 records will fail with error EEOF if the file type does not permit seeking beyond the end of file. For an indexed file, currency is unchanged; a variable-length record sequential file has to be scanned through to find the record number of the last record in the file, and so currency on failure is at EOF.

ERROR CODES

flErrno may take the following values:

[EABORTED]	The last FlAP function was terminated by *flqAbort*.
[EALLOK]	The last FlAP function completed without error.
[EBADFD]	The given file descriptor does not correspond to an open file.
[EBADINIT]	*flInit* failed; probably due to either lack of buffer space or transport service failing to initialise.
[EBADKEY]	The key specification given for this index file is malformed.
[EBADMATCH]	Unknown match type given to *flqFindRec*.
[EBADNAM]	The given file name is not a proper file name.

[EBADOP] Invalid operation for a file of this type.

[EBADORG] The structure of the named file does not correspond to that given (e.g. attempting to open sequential file as an indexed file).

[EBADPAT] The search pattern given to *flqSearch* is malformed.

[EBADPTR] A pointer (to a filename or data buffer) points to outside the process data space.

[EBADRLEN] Record length is too short or too long.

[EBADSIZ] A read or write of a negative number of bytes has been requested.

[EBADSK] Invalid whence parameter to *flqSeek*.

[EBADSTART] *flStart* failed; probably due to not being able to establish communication over transport service.

[EBOF] At the start of the file.

[ECNTBKUP] Can't seek backwards from the end of a variable-length record file.

[ECMPFAIL] *flqCompare* unable to find matching records.

[ECNTDEL] Couldn't delete file.

[ECNTREW] Rewind of a sequential file failed.

[ECNTTRNC] Attempt to truncate file failed.

[ECNTXTND] The file cannot be extended, usually because no space is available on the file system.

[EEOF] Either an attempt is being made to read beyond the end of a file, or to seek beyond the end of the file when this is not permitted for the file type.

[EFLNTFND] No file with that name can be found.

[EIDXSPC] Failure allocating index space for new entry to file.

[EINTRNL] Ghastly internal error in FIAP. Usually a problem allocating memory. If this

error should occur, your file *may* have been corrupted.

[ELINKDOWN] The transport service has failed.

[ENOCURRENTOP] *flDone* has been called and no FlAP operation is in progress.

[ENODUPS] Duplicate keys not allowed in this indexed file. The index and file are reset to their state before the record operation started.

[ENOIDX] No such index.

[ENOKEY] No primary key specified for this indexed file.

[ENOORG] No such file organisation is supported.

[ENOTFOUND] Search requested by *flqSearch* has failed to find a record matching the pattern given.

[ENOTSTARTED] *flStart* must be called successfully before FlAP operations can commence.

[EOPACTIVE] A FlAP operation is already active; only one operation at a time is permitted.

[EOSERR] Miscellaneous operating system error not covered by the above.

[ERECBAD] Error reading a record (usually because end of file occured before the end of the record).

[ETOMNYFLS] Too many files open.

[ETOMNYKEYS] Too many keys defined for index file.

NOTES

flErrno is **always** set by a call to a FlAP function.

Error code constants are defined in *flap.h*.

NAME

Filetypes – available FlAP file types

SYNOPSIS

#include "flap.h"

DESCRIPTION

Currently 11 file types are defined for FlAP. They are classified into 4 record types as follows:

Fixed-length records are records which occupy a fixed amount of space per record. Sequential (*FL_SEQ*), Relative (*FL_REL*), and Indexed (*FL_IND*) files are available. There is no distinction between FL_SEQ and FL_REL files in the current Unix server implementation.

Variable-length records are records which occupy a varying amount of space per record. The normal variable-length record format, format 1, has the following file types available: Sequential (*VL1_SEQ*), Relative (*VL1_REL*), and Indexed (*VL1_IND*). An alternative variable-length record format, format 2, is available for sequential files only (*VL2_SEQ*). This format complies with American National Standard X3.27-1977, and should be used when creating conforming files for tape transfer.

Text records are variable-length records whose sequential format corresponds to that for text files under Unix. Two file types are available, Sequential (*TXT_SEQ*), and Relative (*TXT_REL*). Relative text files offer *flqAddRec*(FlAP) and *flqDeleteRec*(FlAP) facilities more appropriate to text handling, at some expense in disc space and processing time. Their use should be restricted to temporary files into which sequential text files are read for processing and the text copied back to a sequential file on termination. Currently, they are wasteful of disc space (extending a record allocates fresh data space for the record, and the unused space is never reclaimed) and very fragile (see *Intro*(FlAP)).

Bytestream files have no record structure at all. Multiple bytes may be read or written in one operation, and *flqSeek*(FlAP) deals in byte offsets. Sequential (*STREAM_SEQ*) and Relative (*STREAM_REL*) file types are available, though there is no distinction between the two under the current Unix server implementation.

NOTES

Under the current Unix server, the normal variable-record format, format 1, is to preface the record itself

with a two-byte binary count of the bytes in the record itself (not including the count preface), the count being stored low-byte – high-byte. This count is invisible to the record-processing operations. Note that relative variable-length files must have a maximum record length specified when the file is opened, and records are stored in fixed-size cells big enough to hold a maximum-length record. The maximum record length specified should not include space for the record count prefix; FlAP handles all such matters internally.

Variable-length record format 2 prefaces each record with a 4–byte decimal count of the record length, inclusive of the count. FlAP record processing operations adjust the record length to that of just the record data.

The file types and operations are supplied by URAT; hence the above is essentially abstracted from the relevant URAT manual page.

SEE ALSO

Recordspec(FlAP)

NAME

Recordspec – record details specification

SYNOPSIS

```
#include "fiap.h"

struct flKeySeg
{
        u_short start ;
        u_short length ;
}

struct flKeySpec
{
        u_short flags ;
        u_short no_segs ;
        struct flKeySeg  seg [MAXSEGS] ;
}

struct flRecSpec
{
        u_short reclen ;
        u_short no_keys ;
        struct flKeySpec *kspec ;
}
```

DESCRIPTION

The record length of a fixed-length record file is given in *reclen*. For a variable-length record file *reclen* specifies the maximum record length permissible. This is mandatory for a relative file, but optional for sequential and indexed files and may be set to 0 to indicate no maximum record length for VL1_SEQ, VL2_SEQ, TXT_SEQ and TXT_REL files. A non-zero record length specified for a bytestream file

(STREAM_SEQ, STREAM_REL) just imposes an upper limit on the number of bytes that may be read during one *flqGetRec*(FlAP) operation.

Indexed files are indexed on one *primary key*. They may also be indexed on further alternate keys, up to a maximum of *MAXKEYS* keys (including the *primary key*). *no_keys* gives the number of keys on which to index, and must be a minimum of 1 (the *primary key*).

A *key* is made up from up to *MAXKEYSIZE* printable ASCII characters (i.e. it may not contain control characters). The format of each key is specified by the corresponding entry in an array of key specifications pointed to by *kspec* (so the primary key specification is at kspec[0], the next key specification at kspec[1] etc.). Up to *MAXKEYS* keys are supported.

A key is made up from 1 to *MAXSEGS* segments, a segment being a number of bytes from the record starting at a given offset. Hence a key may be made up from several different parts of the record. For fixed-length records and variable-length records for which a maximum record length is given, each segment must lie wholly inside the record. Should a variable-length record not be long enough to wholly or partially contain the segment, the key value for that segment will be padded with spaces. Similarly, if the record contains non-printable characters these are replaced in the key with spaces.

flags should be set to either *DUPS_OK* or *NO_DUPS* to indicate whether duplicate keys should be permitted, or whether any attempt to add a record with a duplicate key should be rejected. Forbidding duplicate keys saves some disc space for the index; allowing duplicate keys restricts maximum key size to *MAXKEYSIZE* - 3 characters, as 3 characters are automatically added on to the key to ensure duplicate keys are kept separate).

NOTES

Using the Unix server, the current value of *MAXKEYS* is 8, the current value of *MAXSEGS* is also 8, and the current value of *MAXKEYSIZE* is 80.

SEE ALSO

Filetypes(FlAP)

NAME

start – FlAP application starter

SYNOPSIS

start [**-arguments**]

DESCRIPTION

start allows FlAP applications to be started from a Unix command line if used with an appropriate terminal emulator such as VT100 (described elsewhere). It does this by sending a special FlAP starter sequence. If this is responded to correctly, it sends the name it was run under and any arguments to the microcomputer and runs the FlAP server, passing any arguments to the FlAP server.

USE

start is normally used by renaming it to the name of the particular FlAP application. Invoking it under that name will cause it to attempt to start a microcomputer program with the same name and, if successful, to start the FlAP server.

Multiple FlAP applications can use one copy of *start* by creating appropriately-named links to it with *ln*(1).

Because the current PC FlAP interface uses the SERINT serial port handler, the most convenient way to use FlAP applications is with the VT100 terminal emulator that accompanies SERINT. This must be invoked with the TERMINAL program, which handles loading VT100 and any requested FlAP application.

SEE ALSO

Unix-Server(FlAP), *ln*(1), *FlAP Application Writer's Guide*

NAME

flap – Unix FlAP server

SYNOPSIS

flap [–d[d[d..]]]

DESCRIPTION

flap is a FlAP server for 4.3BSD Unix which uses MMMS and URAT to provide a FlAP service to the current terminal. It should be started before the microcomputer application.

USE

flap recognises one argument, requesting various levels of debugging information be recorded in a file *flap.log*. Any other arguments are ignored.

-d FlAP functions used.

-dd FlAP functions used and packet flow.

-ddd FlAP functions used, packet flow and packet contents.

If more than three levels of debugging are requested, FlAP passes on the excess to MMMS (see *MS_Init*, MMMS Programmer's Manual).

When closed down, *flap* waits for two seconds before exiting. This is to allow time for the microcomputer terminal emulator to be restarted.

SEE ALSO

MMMS Programmer's Manual, *URAT Programmer's Manual*

NAME

Vectorspec – record vector specification

SYNOPSIS

```
#include "flap.h"

struct flRecVector
{
        long    offset ;
        u_short whence ;
        u_short bufsiz ;
        u_short reclen ;
        char    *bufptr ;
} ;
```

DESCRIPTION

flqVAdd, *flqVGet* and *flqVUpdate* all operate on not one record but a vector of records. This is an array of record buffer details of type *struct flRecVector*.

Each vector operation is equivalent to a series of (Seek, Add/Update/Get) pairs, with the parameters to the Seek being *offset* and *whence*.

The actual record data is either read from or written to *bufptr*. *reclen* is the number of bytes that may be read from *bufptr*, used by *flqVAdd* and *flqVUpdate*, and set by *flqVGet*. *bufsiz* is the maximum number of bytes that can be written to *bufptr*, and is used by *flqVGet*.

For example, to read 10 records using *flqVGet*, an array of 10 elements of type *struct flRecVector* must be set up, and for each element buffer space of some appropriate length allocated, *bufsiz* set to that length and *bufptr* set to point to the buffer. Then a call to *flqVGet* may be issued. When this completes, each buffer at *bufptr* will contain *reclen* bytes of data, unless *reclen* was greater than *bufsiz* in which case it will contain *bufsiz* bytes. In either case the length of the record is placed in *reclen*.

When writing records using either *flqVAdd* or *flqVUpdate*, *bufptr* should be set to point at the record data, and *reclen* set to the number of bytes of record data available.

SEE ALSO

flqVAdd(FlAP), *flqVGet*(FlAP), *flqVUpdate*(FlAP), *Filetypes*(FlAP), *Recordspec*(FlAP)

NAME

flqAbort – queue abort request for ongoing FlAP operation

SYNOPSIS

#include "flap.h"

int flqAbort()

DESCRIPTION

flqAbort queues an abort request for the ongoing FlAP operation. *flDone*(FlAP) should then be used to wait for the end of the operation.

Because it must always be possible to get an abort request to the notice of the system hosting the files, multiple FlAP operation requests cannot be queued at once. It must be admitted this is somewhat of a convenience measure for this implementation, and the restriction could go away in later versions.

Aborting an operation will only have an effect if the operation is a time-consuming one, e.g. reading/writing records, searching etc.

RETURN VALUE

The value –1 is returned if there is no outstanding FlAP operation or there has been a failure of the communication link. Otherwise 0 is returned.

ERRORS

flqAbort will fail for any of the following reasons:

[EINTRNL]	Internal error in FlAP.
[ELINKDOWN]	The FlAP communication link has failed.
[ENOCURRENTOP]	No FlAP operation is currently active.

NAME

flqAddRec – queue request to add a record

SYNOPSIS

#include "flap.h"

int flqAddRec(fd, bufsiz, bufptr)

int fd, bufsiz ;

char *bufptr ;

DESCRIPTION

flqAddRec queues a request to add a new record to the file with descriptor *fd* using up to *bufsiz* bytes from *bufptr*.

If the file is of fixed-length records the record will be either truncated or padded with ASCII NULs if *bufsiz* does not correspond to the record length.

If the file is of variable-length records the record will be truncated if it exceeds the maximum record length specified.

In most cases, *flqAddRec* merely locates the current end of file and adds a record.

In the case of indexed files, the new record is added to the data file and all indexes.

For relative text (TXT_REL) files, the new record is inserted into the file *before* the current record. All succeeding records 'move down'; if a record is inserted before record *n*, record *n* becomes record *n*+1.

If the file is a bytestream, *bufsiz* bytes are written, starting at EOF.

CURRENCY

The current record becomes the record after the added record; this will be EOF for all but TXT_REL and indexed files. For indexed files, currency is the record following the added record in the current index.

NOTES

In the case of indexed files, all indexes are updated to reflect the addition of the data record. No provision is made for delaying updating of indexes not currently in use.

RETURN VALUE

If the request is queued properly, 0 is returned. Otherwise –1 is returned.

The result given by *flDone*(FlAP) is the number of bytes written to the new record if successful, otherwise –1 and an error code.

SEE ALSO

flqDelRec(FlAP), *flqUpdateRec*(FlAP)

ERRORS

flqAddRec will fail and the record will not be added for any of the following reasons:

[EBADFD]	The given file descriptor does not correspond to an open file.
[EBADPTR]	*bufptr* is not a valid pointer.
[EBADSIZ]	*bufsiz* is negative.
[ECNTXTND]	Cannot extend file to make room for new record.
[EIDXSPC]	No space to add index entry.
[EINTRNL]	Internal error in FlAP.
[ELINKDOWN]	The transport service has failed.
[ENODUPS]	Attempting to duplicate a key in an indexed file which does not permit duplicate keys.
[ENOTSTARTED]	No FlAP operation can be started until *flStart* has been called successfully.
[EOPACTIVE]	A FlAP operation is currently active.
[EOSERR]	Miscellaneous error from operating system hosting the file handler not covered by the above.
[ERECBAD]	Bad record found whilst looking for end of file.

NAME

flqVAdd – queue request to add a vector of records

SYNOPSIS

#include "flap.h"

int flqVAdd(fd, no_recs, recv)

int fd, no_recs ;

struct flRecVector *recv ;

DESCRIPTION

flqVAdd queues a request to add up to *no_recs* new records from *recv* to the file with descriptor *fd*.

This is equivalent to multiple calls to *flqSeek*(FlAP) and *flqAddRec*(FlAP), but is more efficient (separate *flqSeek* and *flqAddRec* operations must wait for confirmation from the file server before permitting the next seek or add operation to start).

RETURN VALUE

If the request is queued properly, 0 is returned. Otherwise –1 is returned.

If *n* records are successfully added without error, the result given by *flDone*(FlAP) is *n*; if an error occurs adding record *n* the result is *–n* and an error code.

SEE ALSO

flqDelRec(FlAP), *flqVUpdate*(FlAP), *Vectorspec*(FlAP)

ERRORS

flqVAdd will fail and the records will not be added for any of the following reasons:

[EBADFD]	The given file descriptor does not correspond to an open file.
[EBADPTR]	An entry in *recv* does not have *bufptr* pointing to a valid buffer.
[EBADSIZ]	Either *no_recs* is less than 1, or one entry for *reclen* in *recv* is not in the range 0 to MAXRECLEN.

[EBADSK] An entry in *recv* has an invalid value for *whence*.

[EBOF] An entry in *recv* attempts to seek to before the start of the file.

[ECNTXTND] Cannot extend file to make room for new record.

[EIDXSPC] No space to add index entry.

[EINTRNL] Internal error in FlAP.

[ELINKDOWN] The transport service has failed.

[ENODUPS] Attempting to duplicate a key in an indexed file which does not permit duplicate keys.

[ENOTSTARTED] No FlAP operation can be started until *flStart* has been called successfully.

[EOPACTIVE] A FlAP operation is currently active.

[EOSERR] Miscellaneous error from operating system hosting the file handler not covered by the above.

[ERECBAD] Bad record found whilst looking for end of file.

NAME

flqClose – queue request to close an open record file

SYNOPSIS

#include "flap.h"

int flqClose(fd)

int fd ;

DESCRIPTION

flqClose attempts to close the open record file with file descriptor *fd*. Any file buffers are written to disc, and *fd* becomes inactive. *flqOpen* must be called before the file can be used again.

If the file was specified as a temporary file, it is automatically deleted on closing.

CURRENCY

Not applicable; currency lost.

RETURN VALUE

The value –1 is returned if an error occurs. 0 is returned if the request is queued properly.

The result given by *flDone*(FlAP) is 0 if the file was closed without error, otherwise –1 and an error code.

ERRORS

flqClose will fail for any of the following reasons:

[EBADFD]	The given file descriptor does not correspond to an open file.
[ECNTXTND]	No room on disc to save index buffers.
[EINTRNL]	Internal error in FlAP.
[ELINKDOWN]	The transport service has failed.
[ENOTSTARTED]	No FlAP operation can be started until *flStart* has been called successfully.
[EOPACTIVE]	A FlAP operation is currently active.

NAME

flqCompare – queue request to compare records

SYNOPSIS

#include "flap.h"

int flqCompare(from_fd, to_fd, no_recs, match)

int from_fd, to_fd, match ;

long no_recs ;

DESCRIPTION

flqCompare compares up to *no_recs* successive records from the file with descriptor *from_fd* to successive records in the file with descriptor *to_fd* starting from the current record in each file. If *match* is FALSE the records are compared until they match; otherwise records are compared until they differ.

If *no_recs* is –1 then records are compared until the end of either file.

CURRENCY

Currency in both files is after the last record compared.

RETURN VALUE

If the request is queued properly, 0 is returned. Otherwise –1 is returned.

If *n* records are successfully compared without error, the result given by *flDone*(FlAP) is *n*; if an error occurs comparing record *n* the result is *–n* and an error code.

ERRORS

flqCompare will fail for any of the following reasons:

[EBADFD]	Either *from_fd* or *to_fd* is an invalid file handle.
[EBADSIZ]	*no_recs* must be greater than 0 or –1.
[EBOF]	Hit the start of either file.
[ECMPFAIL]	No (un)equal records found after *no_recs* records searched.

[EEOF]	Hit the end of either file.
[EINTRNL]	Internal error in FlAP.
[ELINKDOWN]	The FlAP communication link has failed.
[ENOTSTARTED]	No FlAP operation can be started until *flStart* has been called successfully.
[EOPACTIVE]	A FlAP operation is currently active.
[EOSERR]	Miscellaneous error from operating system hosting the file handler not covered by the above.
[ERECBAD]	Corrupt record found. Usually means end of file occurred before expected end of record.

NAME

flqCopy – queue request to copy records

SYNOPSIS

#include "flap.h"

int flqCopy(from_fd, to_fd, no_recs, forwards)

int from_fd, to_fd, forwards ;

long no_recs ;

DESCRIPTION

flqCopy copies up to *no_recs* records from the file with descriptor *from_fd* and adds them to the file with descriptor *to_fd*. Records are copied from the current record onwards. If *forwards* is FALSE and *from_fd* is not a sequential file then records are taken in reverse order from *from_fd*, again starting at the current record.

If *no_recs* is –1 then records are copied until the end or beginning of *from_fd*.

flqCopy is equivalent to multiple *flqGetRec* – *flqAddRec* operations.

CURRENCY

Currency in the from file is the record after the last one copied; in the to file it depends on the file type – see *flqAddRec*(FLAP).

RETURN VALUE

If the request is queued properly, 0 is returned. Otherwise –1 is returned.

If *n* records are successfully copied without error, the result given by *flDone*(FlAP) is *n*; if an error occurs copying record *n* the result is *–n* and an error code.

ERRORS

flqCopy will fail for any of the following reasons:

[EBADFD] Either *from_fd* or *to_fd* is an invalid file handle.

[EBADSIZ]	*no_recs* must be greater than 0 or EOF.
[EBOF]	Hit the start of either file.
[ECNTBKUP]	Couldn't move backwards in from file. Usually due to hitting BOF.
[EEOF]	Hit the end of either file.
[EINTRNL]	Internal error in FlAP.
[ELINKDOWN]	The FlAP communication link has failed.
[ENOTSTARTED]	No FlAP operation can be started until *flStart* has been called successfully.
[EOPACTIVE]	A FlAP operation is currently active.
[EOSERR]	Miscellaneous error from operating system hosting the file handler not covered by the above.
[ERECBAD]	Corrupt record found. Usually means end of file occurred before expected end of record.

NAME

flqCreate – queue request to create a record file

SYNOPSIS

#include "flap.h"

int flqCreate(fname, ftype, spec)

char *name ;

struct RecSpec *spec ;

int ftype ;

DESCRIPTION

flqCreate queues a request to create a record file of the given file type, and specification with the name at *fname*.

If a file of that name already exists it is emptied.

If the modifier TEMPFILE is added to *ftype* the file is placed in temporary storage and deleted when closed.

CURRENCY

At BOF.

NOTES

Valid file types for *ftype* are listed in *Filetypes*(FlAP).

Details of the required record specifications in *spec* are given in *Recordspec*(FlAP).

RETURN VALUE

The value –1 is returned if an error occurs. If the request is queued properly, 0 is returned.

The result given by *flDone*(FlAP) is a non-negative file descriptor if the file is successfully created, otherwise –1 and an error code.

ERRORS

flqCreate will fail and the file will not be created or truncated for any of the following reasons:

[EBADKEY]	Indexed file key specification is invalid — error in number of segments, segment specification or overall key length.
[EBADNAM]	*fname* does not point to a valid filename.
[EBADPTR]	*fname* is not a valid pointer.
[EBADRLEN]	Bad record length given.
[ECNTTRNC]	Existing file could not be truncated. This may be because access to the file is not permitted, or because it is a directory.
[ECNTXTND]	Ran out of disc space trying to write file header.
[EIDXSPC]	No spare memory available for index buffers.
[EINTRNL]	Internal error in FlAP.
[ELINKDOWN]	The transport service has failed.
[ENOKEY]	No primary key specified for indexed file.
[ENOORG]	No such file organisation is supported.
[ENOTSTARTED]	No FlAP operation can be started until *flStart* has been called successfully.
[EOPACTIVE]	A FlAP operation is currently active.
[EOSERR]	Miscellaneous error from operating system hosting the file handler not covered by the above.
[ETOMNYFLS]	You are trying to open too many files at once.
[ETOMNYKEYS]	You can't have more than MAXKEYS indexes for an indexed file.

NAME

flqDelFile – queue request to delete a record file

SYNOPSIS

#include "flap.h"

int flqDelFile(fname)

char *fname ;

DESCRIPTION

flqDelFile queues a request to delete the file with the file name given in *fname*.

If indexes are held in separate files, these are also deleted.

RETURN VALUE

If the request is queued properly, 0 is returned. Otherwise –1 is returned.

The result given by *flDone*(FlAP) is 0 if the file was deleted OK, otherwise –1 and an error code.

SEE ALSO

flqRenFile(FlAP)

ERRORS

flqDelFile will fail and the file will not be deleted (assuming it exists in the first place) for any of the following reasons:

[EBADNAM]	The give file name is not a properly-constructed filename.
[ECNTDEL]	Couldn't delete file — perhaps access permissions not relaxed enough.
[EFLNTFND]	No file with that name could be found.
[EINTRNL]	Internal error in FlAP.
[ELINKDOWN]	The transport service has failed.
[ENOTSTARTED]	No FlAP operation can be started until *flStart* has been called successfully.

[EOPACTIVE] A FIAP operation is currently active.

[EOSERR] Miscellaneous error from operating system hosting the file handler not covered by the above.

NAME

flqDelRec – queue request to delete record(s)

APPLICABLE-FILE-TYPES

FL_IND, VL1_IND, TXT_REL

SYNOPSIS

```
#include "flap.h"

int flqDelRec(fd, no_recs)

int fd ;

long no_recs ;
```

DESCRIPTION

flqDelRec queues a request to delete up to *no_recs* records starting at the current record in the file with descriptor *fd*. If *no_recs* is –1, all records until the end of the file are deleted (this is equivalent to a call to *flqTruncate*).

CURRENCY

The current record becomes the record after the last deleted record.

NOTES

In the case of indexed files, all indexes are updated to reflect the deletion of the data records.

Using the Unix file server, in all but one case the space made available by the deletion can be reused. However, no garbage collection or compaction occurs. In TXT_REL files, the index space may be reused, but not data space.

After *flqDelRec* deletes each record, all records following the deleted record have their record numbers as used by *flqSeek* effectively decremented by one; if record *n* is deleted, records *n*+1 and *n*+2 become records *n* and *n*+1 respectively.

RETURN VALUE

If the request is queued properly, 0 is returned. Otherwise –1 is returned.

If *n* records are successfully deleted without error, the result given by *flDone*(FlAP) is *n*; if an error occurs deleting record *n* the result is *–n* and an error code.

SEE ALSO

flqAddRec(FlAP)

ERRORS

flqDelRec will fail and a record will not be deleted for any of the following reasons:

[EBADFD]	The given file descriptor does not correspond to an open file.
[EBADOP]	Records cannot be deleted in this type of file i.e. file is not of type FL_IND, VL1_IND or TXT_REL.
[EBADSIZ]	*no_recs* must be a positive integer or –1.
[EEOF]	At the end of the file.
[EINTRNL]	Internal error in FlAP.
[ELINKDOWN]	The transport service has failed.
[ENOTSTARTED]	No FlAP operation can be started until *flStart* has been called successfully.
[EOPACTIVE]	A FlAP operation is currently active.
[EOSERR]	Miscellaneous error from operating system hosting the file handler not covered by the above.

NAME

flDone – see if FlAP operation is complete

SYNOPSIS

#include "flap.h"

int flDone(result)

long *result ;

DESCRIPTION

flDone monitors progress of the current FlAP operation.

FlAP operations such as flVAdd, which may transfer considerable amounts of data from the system hosting the files, sometimes rely on frequent calls to *flDone* to transfer data to or from the communication buffers. This is the case for the PC library.

RETURN VALUE

The value –1 is returned if there is no outstanding FlAP operation or there has been a failure of the communication link. If the operation is complete, 0 is returned and **result* set to the value returned by the operation. If the operation not complete, 1 is returned.

SEE ALSO

flWait(FlAP)

ERRORS

flDone will fail for any of the following reasons:

[EINTRNL]	Internal error in FlAP.
[ELINKDOWN]	The FlAP communication link has failed.
[ENOCURRENTOP]	No FlAP operation is currently active.

NAME

flqDuplicate – queue request to duplicate records

APPLICABLE-FILE-TYPES

FL_IND, VL1_IND, TXT_REL, VL1_REL, FL_REL

SYNOPSIS

#include "flap.h"

int flqDuplicate(fd, no_recs, to_offset)

int fd ;

long no_recs, to_offset ;

DESCRIPTION

flqDuplicate duplicates up to *no_recs* records in the file with descriptor *fd*. The duplicated records are added to the file at EOF in the case of VL1_REL and FL_REL files, at offset *to_offset* in the case of TXT_REL files (if *to_offset* is –1, records are added to the end of the file), and just after the record being duplicated in the case of indexed files (note that all indexes **must** be capable of holding duplicate keys).

If *no_recs* is –1, all records from the current record until the end of the file are duplicated.

CURRENCY

Currency in the file is the record after the last duplicate added.

RETURN VALUE

If the request is queued properly, 0 is returned. Otherwise –1 is returned.

If *n* records are successfully duplicated without error, the result given by *flDone*(FlAP) is *n*; if an error occurs duplicating record *n* the result is *–n* and an error code.

ERRORS

flqDuplicate will fail for any of the following reasons:

[EBADFD] *fd* is not a valid file descriptor.

[EBOF]	*to_offset* must be greater than or equal to zero, or –1.
[ECNTXTND]	Cannot extend file to make room for new record.
[EEOF]	At the end of the file — no record to duplicate.
[EIDXSPC]	No space to add index entry.
[EINTRNL]	Internal error in FlAP.
[ELINKDOWN]	The FlAP communication link has failed.
[ENODUPS]	Attempting to duplicate a key in an indexed file which does not permit duplicate keys.
[ENOTSTARTED]	No FlAP operation can be started until *flStart* has been called successfully.
[EOPACTIVE]	A FlAP operation is currently active.
[ERECBAD]	Bad record found whilst looking for end of file.

NAME

flqFindRec – queue request to find a record in an indexed file

APPLICABLE-FILE-TYPES

FL_IND, VL1_IND

SYNOPSIS

#include "flap.h"

int flqFindRec(fd, index_no, match_type, keyoffset, keysiz, bufsiz, bufptr)

int fd, index_no, match_type, bufsiz ;

char *bufptr ;

DESCRIPTION

flqFindRec works in the same manner as *flqGetRec*, except that rather than reading the current record, it first searches for a match to a key constructed from *keysiz* bytes starting at *keyoffset* in the record at *bufptr*, and returns that record in *bufptr* if a suitable one is found.

index_no specifies the index which is to be searched (request index 0 to search the current index). The selected index remains the current index.

Several match criterion are available via *match_type*.

EXACT	Find the first record with a key exactly matching the given key.
PARTIAL	Find the first record with a key greater than or equal to the given key.
GREATER	Find the first record with a key greater than the given key.
NOSEARCH	Don't find a record, just switch index. Do not even return the current record in that index, just 0 if successful.

CURRENCY

If the find succeeds, the current record is the one after the find, unless the NOSEARCH option was requested, in which case the currency is unchanged. The current record number will be DONTKNOW. If the find fails, the current record is unchanged.

NOTES

In indexes which permit duplicate keys, the oldest duplicate is found. The duplicates can then be stepped through in the order in which they were added using *flqGetRec*.

RETURN VALUE

If the request is queued properly, 0 is returned. Otherwise –1 is returned.

The result given by *flDone*(FlAP) is the number of bytes transferred to *bufptr* if the operation was successful, otherwise –1 and an error code.

SEE ALSO

flqGetRec(FlAP)

ERRORS

flqFindRec will fail and the record will not be located for any of the following reasons:

[EBADFD]	The given file descriptor does not correspond to an open file.
[EBADMATCH]	Unknown match type specified.
[EBADOP]	Find cannot be used for any file not an indexed file, that is of type FL_IND or VL1_IND.
[EBADSIZ]	Either *bufsiz*, *keysiz* or *keyoffset* is invalid, either less than 0 or greater than MAXRECLEN.
[EEOF]	Find failed — no suitable match (i.e. hit end of index without search succeeding. Currency is as before call to *flqFindRec*).
[EINTRNL]	Internal error in FlAP.
[ELINKDOWN]	The transport service has failed.
[ENOIDX]	No such index.
[ENOTSTARTED]	No FlAP operation can be started until *flStart* has been called successfully.
[EOPACTIVE]	A FlAP operation is currently active.

[EOSERR] Miscellaneous error from operating system hosting the file handler not covered by the above.

NAME

flqGetRec – queue request to read a record

SYNOPSIS

#include "flap.h"

int flqGetRec(fd, bufsiz, bufptr)

int fd, bufsiz ;

char *bufptr ;

DESCRIPTION

flqGetRec queues a request to read the current record in the file with descriptor *fd* into the destination indicated by *bufptr*. Up to *bufsiz* bytes are written to the destination buffer — if the record is longer the remaining bytes are ignored. The length of the record is returned.

If *bufptr* is NULL or *bufsiz* is zero then no data is transferred, but the record length is still returned.

CURRENCY

The next record becomes the current record.

NOTES

Unexpectedly long variable-length records could cause the destination buffer to overflow, hence the need for *bufsiz*.

RETURN VALUE

If the request is queued properly, 0 is returned. Otherwise –1 is returned.

The result given by *flDone*(FlAP) is the number of bytes in the record if *flqGetRec* was successful, otherwise –1 and an error code.

ERRORS

flqGetRec will fail and the record will not be retrieved for any of the following reasons:

[EBADFD] The given file descriptor does not correspond to an open file.

[EBADPTR]	*bufptr* is not a valid pointer.
[EBADSIZ]	*bufsiz* is negative.
[EEOF]	At the end of the file — no record to read.
[EINTRNL]	Internal error in FlAP.
[ELINKDOWN]	The transport service has failed.
[ENOTSTARTED]	No FlAP operation can be started until *flStart* has been called successfully.
[EOPACTIVE]	A FlAP operation is currently active.
[EOSERR]	Miscellaneous error from operating system hosting the file handler not covered by the above.
[ERECBAD]	Record is corrupt. Usually means end of file was found before the expected end of record.

NAME

flqVGet – queue request to read a vector of records

SYNOPSIS

#include "flap.h"

int flqVGet(fd, no_recs, recv)

int fd, no_recs ;

struct flRecVector *recv ;

DESCRIPTION

flqVGet queues a request to read up to *no_recs* records specified in *recv* from the file with descriptor *fd*.

This is equivalent to multiple calls to *flqSeek*(FlAP) and *flqGetRec*(FlAP), but is more efficient (separate *flqSeek* and *flqGetRec* operations must wait for confirmation from the file server before permitting the next seek or get operation to start).

RETURN VALUE

If the request is queued properly, 0 is returned. Otherwise –1 is returned.

If *n* records are successfully read without error, the result given by *flDone*(FlAP) is *n*; if an error occurs reading record *n* the result is *–n* and an error code.

SEE ALSO

flqVAdd(FlAP), *flqVUpdate*(FlAP), *Vectorspec*(FlAP)

ERRORS

flqVGet will fail and the records will not be retrieved for any of the following reasons:

[EBADFD]	The given file descriptor does not correspond to an open file.
[EBADPTR]	An entry in *recv* does not have *bufptr* pointing to a valid buffer.
[EBADSIZ]	Either *no_recs* is less than 1, or one entry for *reclen* in *recv* is not in the range 0 to MAXRECLEN.

[EBADSK]	An entry in *recv* has an invalid value for *whence*.
[EBOF]	An entry in *recv* attempts to seek to before the start of the file.
[EINTRNL]	Internal error in FlAP.
[ELINKDOWN]	The transport service has failed.
[ENOTSTARTED]	No FlAP operation can be started until *flStart* has been called successfully.
[EOPACTIVE]	A FlAP operation is currently active.
[EOSERR]	Miscellaneous error from operating system hosting the file handler not covered by the above.
[ERECBAD]	Record is corrupt. Usually means end of file was found before the expected end of record.

NAME

flInit – establish FlAP communication

SYNOPSIS

#include "flap.h"

int flInit()

int flStart()

int flReady()

DESCRIPTION

flInit initialises the FlAP communication link and various miscellaneous internal FlAP settings.

flStart commences attempts to establish communication with the FlAP file server.

flReady indicates whether or not communication has been established.

Note that *flShutdown* **must always** be called before the application exits.

NOTES

The usual sequence to be followed when attempting to establish a link to the file server is to call *flInit*, then *flStart*, and then loop on *flReady* until either the link is established or an error occurs. The functions are split in this fashion to allow the application control while communication is being established; if a failure does occur, it may not be signalled until the appropriate timeout has expired, which could be a significant period of time.

RETURN VALUE

flInit and *flStart* return 0 if successful, otherwise –1 and an error code.

flReady returns 1 if the communication link is established, 0 if it isn't, and –1 and an error code in the event of an error.

SEE ALSO

flShutdown(FlAP)

ERRORS

flInit, *flStart* and *flReady* will fail for any of the following reasons:

[EBADINIT] *flInit* failed; probably due to either lack of buffer space or transport service failing to initialise.

[EBADTSTART] *flStart* failed; probably due to transport service being unable to establish communication.

[EINTRNL] Internal error in FlAP.

[ELINKDOWN] The FlAP communication link has failed.

[ENOTSTARTED] *flReady* called before *flStart*.

NAME

flqOpen – queue request to open an existing record file

SYNOPSIS

#include "flap.h"

int flqOpen(fname, ftype, reclen)

char *fname ;

int ftype, reclen ;

DESCRIPTION

flqOpen queues a request to attempt to open a record file of the given file type using the name given in *fname*. In cases where the record length *reclen* is required and cannot be deduced from the file (FL_SEQ, FL_REL, VL1_REL) it must be supplied — otherwise *reclen* is ignored.

CURRENCY

At BOF.

NOTES

Valid file types for *ftype* are listed in *Filetypes*(FlAP). After opening, the current record is the first record in the file.

RETURN VALUE

The value –1 is returned if an error occurs. If the request is queued properly, 0 is returned.

The result given by *flDone*(FlAP) is a non-negative file descriptor if the file was opened without error, otherwise –1 and an error code.

ERRORS

flqOpen will fail and the file will not be opened for any of the following reasons:

[EBADNAM] *fname* does not point to a valid filename.

[EBADORG] File is not of the organisation specified.

[EBADPTR]	*fname* is not a valid pointer.
[EBADRLEN]	Given record length is invalid.
[EFLNTFND]	File (either data file or index file, if appropriate) not found.
[EINTRNL]	Internal error in FlAP.
[ELINKDOWN]	The transport service has failed.
[ENOORG]	No such file organisation is supported.
[ENOTSTARTED]	No FlAP operation can be started until *flStart* has been called successfully.
[EOPACTIVE]	A FlAP operation is currently active.
[EOSERR]	Miscellaneous error from operating system hosting the file handler not covered by the above.
[ETOMNYFLS]	You are trying to open too many files at once.

NAME

flqRenFile – queue request to rename a record file

SYNOPSIS

#include "flap.h"

int flqRenFile(from_fname, to_fname)

char *from_fname, *to_fname ;

DESCRIPTION

flqRenFile queues a request to rename the file with the file name given in *from_fname* to the new name given in *to_fname*.

If indexes are held in separate files, these are renamed too.

NOTES

Any existing file with a name of *to_fname* will be deleted. Watch it.

RETURN VALUE

If the request is queued properly, 0 is returned. Otherwise –1 is returned.

The result given by *flDone*(FlAP) is 0 if the renaming was successful, otherwise –1 and an error code.

SEE ALSO

flqDelFile(FlAP)

ERRORS

flqRenFile will fail and the file will not be renamed (assuming it exists in the first place) for any of the following reasons:

[EBADNAM]	One of the given file names is not a properly-constructed filename.
[ECNTDEL]	Couldn't rename file — perhaps access permissions not relaxed enough.
[EFLNTFND]	No file with that name could be found.
[EINTRNL]	Internal error in FlAP.

[ELINKDOWN]	The transport service has failed.
[ENOTSTARTED]	No FlAP operation can be started until *flStart* has been called successfully.
[EOPACTIVE]	A FlAP operation is currently active.
[EOSERR]	Miscellaneous error from operating system hosting the file handler not covered by the above.

NAME

flqSearch – queue request to search for a record

SYNOPSIS

#include "flap.h"

int flqSearch(fd, regex, no_recs, forwards, bufsiz, bufptr)

int fd, bufsiz ;

char *regex, *bufptr ;

long no_recs ;

DESCRIPTION

flqSearch queues a request to search up to *no_recs* beginning with the current record in the file with descriptor *fd* for a record whose contents match the regular expression given by *regex*. The first matching record is returned; up to *bufsiz* bytes are written to the destination buffer at *bufptr* - if the record is longer the remaining bytes are **silently** discarded. If *no_recs* is –1, the search continues until the beginning or end of the file, depending on the direction of the search.

If *bufptr* is NULL or *bufsiz* is zero then no data is transferred, but the search still takes place.

The regular expression is a standard Unix regular expression, as used in *ed*(1), *grep*(1) etc. *regex* is a NUL-terminated string.

CURRENCY

The next record after the last one compared becomes the current record.

NOTES

Unexpectedly long variable-length records could cause the destination buffer to overflow, hence the need for *bufsiz*.

RETURN VALUE

If the request is queued properly, 0 is returned. Otherwise –1 is returned.

If *n* records are successfully searched without error, the result given by *flDone*(FlAP) is *n*; if an error occurs searching record *n* the result is *–n* and an error code. If all requested records are searched and no matching record is found, *–n* and an error code of ENOTFOUND is returned.

SEE ALSO

regexp(3)

ERRORS

flqSearch will fail and the record will not be retrieved for any of the following reasons:

[EBADFD]	The given file descriptor does not correspond to an open file.
[EBADPAT]	The string pointed to by *regex* does not form a valid regular expression; it may simply be too long.
[EBADPTR]	*bufptr* is not a valid pointer.
[EBADSIZ]	*bufsiz* is negative.
[EBOF]	Hit the start of file.
[ECNTBKUP]	Couldn't move backwards in file.
[EEOF]	Hit the end of the file.
[EINTRNL]	Internal error in FlAP.
[ELINKDOWN]	The transport service has failed.
[ENOTFOUND]	Search failed.
[ENOTSTARTED]	No FlAP operation can be started until *flStart* has been called successfully.
[EOPACTIVE]	A FlAP operation is currently active.
[EOSERR]	Miscellaneous error from operating system hosting the file handler not covered by the above.
[ERECBAD]	Record is corrupt. Usually means end of file was found before the expected end of record.

NAME

flqSeek – queue request to move to a record

SYNOPSIS

#include "flap.h"

long flqSeek(fd, offset, whence)

int fd, whence ;

long offset ;

DESCRIPTION

flqSeek queues a request to move to the record indicated by *offset* and *whence*, and makes that record the current record.

offset is the number of records away from the starting point given by *whence*, and may be positive or negative; *whence* may be one from

0 *offset* is relative to the start of the file.

1 *offset* is relative to the current position.

2 *offset* is relative to the end of the file.

The new current record number is returned. This may be unknowable, in which case DONTKNOW is returned; this occurs in indexed and variable-length sequential files only. For all variable-length sequential file types, DONTKNOW is **only** returned as the result of a seek to the end of the file (as it is not possible to know the number of records skipped without scanning through them). Indexed file currency may be set to DONTKNOW after an *flqAddRec* or *flqFindRec*.

Record numbers run from 0 upwards.

CURRENCY

The new record becomes the current record.

NOTES

Limitations on *flqSeek* vary depending upon the type of file in use.

Sequential files: Attempts to seek beyond the end of the file will be curtailed at the end of the file. Using *flqSeek* with an offset relative to the end of the file on variable-length record files can only be used if that offset is zero, as it is not possible to know how many records there are in the file and thus how far to seek. Similarly, you can seek to the end of a variable-length file, but as the number of records passed over is not known the current record number is set to DONTKNOW. Note that seeking through variable-length records can only be accomplished by reading each and every intervening record, and seeking backwards can only be accomplished by rewinding the file and reading through forwards.

Relative files: *flqSeek* cannot be used to extend the file (by seeking beyond the end-of-file) for relative text files (TXT_REL).

Indexed files: *flqSeek* operates on indexed files by stepping through the current index (possibly first moving to the start or end of the index). The current record number may well be unknowable (e.g. after a *flqFindRec*(FlAP), or a *flqAddRec*(FlAP)), in which case *flqSeek* returns a value of DONTKNOW. Again, any attempt to *flqSeek* beyond the end of the file is curtailed at the end of the file.

RETURN VALUE

If the request is queued properly, 0 is returned. Otherwise –1 is returned.

The result given by *flDone*(FlAP) is the new current record number (possibly DONTKNOW for indexed files) if *flqSeek* was successful, otherwise –1 and an error code.

ERRORS

flqSeek will fail for any of the following reasons:

[EBADFD]	The given file descriptor does not correspond to an open file.
[EBADSK]	*whence* is invalid.
[EBOF]	Can't seek backwards from the start of the file.
[ECNTBKUP]	Can't use negative offsets from the end of a variable-length record sequential file.

[ECNTREW]	Couldn't rewind file to its start.
[EEOF]	Can't move beyond end of file for this file type.
[EINTRNL]	Internal error in FlAP.
[ELINKDOWN]	The transport service has failed.
[ENOTSTARTED]	No FlAP operation can be started until *flStart* has been called successfully.
[EOPACTIVE]	A FlAP operation is currently active.
[EOSERR]	Miscellaneous error from operating system hosting the file handler not covered by the above.
[ERECBAD]	Corrupt record encountered whilst scanning forward through file. Usually end of file has occurred before the expected end of record.

NAME

flShutdown – close down the FlAP communication channel

SYNOPSIS

#include "flap.h"

void flShutdown()

DESCRIPTION

flShutdown closes down the communication link used by FlAP. It **must always** be called before any application using FlAP exits, otherwise debris may be left behind which would cause a crash (this **will** be the case on any MS-DOS machine).

NAME

flqTruncate – queue request to truncate the file

SYNOPSIS

#include "flap.h"

int flqTruncate(fd)

int fd ;

DESCRIPTION

flqTruncate queues a request to delete the current record onwards in the file with descriptor *fd* and truncate the file.

CURRENCY

The current record becomes EOF. In cases of failure the current record is undisturbed.

RETURN VALUE

If the request is queued properly, 0 is returned. Otherwise –1 is returned.

The result given by *flDone*(FlAP) is 0 if the file is truncated successfully, otherwise –1 and an error code.

SEE ALSO

flqDelRec(FlAP)

ERRORS

flqTruncate will fail and records will not be deleted for any of the following reasons:

[EBADFD]	The given file descriptor does not correspond to an open file.
[ECNTTRNC]	Couldn't truncate the data file.
[EEOF]	At the end of the file.
[EINTRNL]	Internal error in FlAP.
[ELINKDOWN]	The transport service has failed.
[ENOTSTARTED]	No FlAP operation can be started until *flStart* has been called successfully.

[EOPACTIVE] A FIAP operation is currently active.

[EOSERR] Miscellaneous error from operating system hosting the file handler not covered by the above.

NAME

flqUpdateRec – queue request to update a record

SYNOPSIS

#include "flap.h"

int flqUpdateRec(fd, bufsiz, bufptr)

int fd, bufsiz ;

char *bufptr ;

DESCRIPTION

flqUpdateRec queues a request to update the current record in the file with descriptor *fd* using up to *bufsiz* bytes from *bufptr*.

If the file is of fixed-length records the record will be either truncated or padded with ASCII NULs if *bufsiz* does not correspond to the record length.

If the file is of variable-length records the record will be truncated if it exceeds the maximum record length specified.

If *flqUpdateRec* is called when the current record is EOF, it adds a new record to the file (at EOF *flqUpdateRec* is equivalent to *flqAddRec*).

CURRENCY

The current record becomes the record after the updated record. In the case of indexed files, this is the record following the record to be updated **before** updating starts; the updating process may well change the key, and thus the position of the updated record in the file.

NOTES

It is possible to use *flqUpdateRec* on sequential variable-length record files, but the practice is heavily discouraged. Unless the updated record is the same length as the old record, the rest of the file will become hopelessly corrupted.

RETURN VALUE

If the request is queued properly, 0 is returned. Otherwise –1 is returned.

The result given by *flDone*(FlAP) is the number of bytes written to the file if *flqUpdateRec* is successful, otherwise –1 and an error code.

SEE ALSO

flqSeek(FlAP), *flqAddRec*(FlAP)

ERRORS

flqUpdateRec will fail and the record will not be updated for any of the following reasons:

[EBADFD]	The given file descriptor does not correspond to an open file.
[EBADPTR]	*bufptr* is not a valid pointer.
[EBADSIZ]	*bufsiz* is negative or exceeds MAXRECLEN.
[ECNTXTND]	Can't extend file to make room for new record.
[EIDXSPC]	No space in index to add new key entries (on updating, new keys are added before old ones are deleted).
[EINTRNL]	Internal error in FlAP.
[ELINKDOWN]	The transport service has failed.
[ENODUPS]	Update is attempting to duplicate a key in an indexed file which does not permit duplicate keys.
[ENOTSTARTED]	No FlAP operation can be started until *flStart* has been called successfully.
[EOPACTIVE]	A FlAP operation is currently active.
[EOSERR]	Miscellaneous error from operating system hosting the file handler not covered by the above.

NAME

flqVUpdate – queue request to update a vector of records

SYNOPSIS

#include "flap.h"

int flqVUpdate(fd, no_recs, recv)

int fd, no_recs ;

struct flRecVector *recv ;

DESCRIPTION

flqVUpdate queues a request to rewrite up to *no_recs* records with data from *recv* to the file with descriptor *fd*.

This is equivalent to multiple calls to *flqSeek*(FlAP) and *flqUpdateRec*(FlAP), but is more efficient (separate *flqSeek* and *flqUpdateRec* operations must wait for confirmation from the file server before permitting the next seek or update operation to start).

RETURN VALUE

If the request is queued properly, 0 is returned. Otherwise –1 is returned.

If *n* records are successfully updated without error, the result given by *flDone*(FlAP) is *n*; if an error occurs updating record *n* the result is *–n* and an error code.

SEE ALSO

flqVAdd(FlAP), *flqVGet*(FlAP)

ERRORS

flqVUpdate will fail and the records will not be added for any of the following reasons:

[EBADFD] The given file descriptor does not correspond to an open file.

[EBADPTR] An entry in *recv* does not have *bufptr* pointing to a valid buffer.

[EBADSIZ] Either *no_recs* is less than 1, or one entry for *reclen* in *recv* is not in the range 0 to MAXRECLEN.

[EBADSK]	An entry in *recv* has an invalid value for *whence*.
[EBOF]	An entry in *recv* attempts to seek to before the start of the file.
[ECNTXTND]	Cannot extend file to make room for new record.
[EIDXSPC]	No space to add index entry.
[EINTRNL]	Internal error in FlAP.
[ELINKDOWN]	The transport service has failed.
[ENODUPS]	Attempting to duplicate a key in an indexed file which does not permit duplicate keys.
[ENOTSTARTED]	No FlAP operation can be started until *flStart* has been called successfully.
[EOPACTIVE]	A FlAP operation is currently active.
[EOSERR]	Miscellaneous error from operating system hosting the file handler not covered by the above.
[ERECBAD]	Bad record found whilst looking for end of file.

NAME

flWait – wait for FlAP operation to complete

SYNOPSIS

#include "flap.h"

long flWait()

DESCRIPTION

flWait monitors progress of the current FlAP operation until it has completed.

RETURN VALUE

The value –1 is returned if there is no outstanding FlAP operation or there has been a failure of the communication link. When the operation is complete, the value returned by the operation is returned.

SEE ALSO

flDone(FlAP)

ERRORS

flWait will fail for any of the following reasons:

[EINTRNL]	Internal error in FlAP.
[ELINKDOWN]	The FlAP communication link has failed.
[ENOCURRENTOP]	No FlAP operation is currently active.

Appendix D

MMMS Programmer's Manual

This page intentionally left blank

NAME

Intro – introduction to MMMS library facilities

DESCRIPTION

MMMS (**Micro-M**ainframe **M**essage **S**ervice) is a protocol which provides a reliable bi-directional transport service between applications running on computers (typically a micro and a mainframe) connected by an asynchronous terminal line.

Applications wishing to use MMMS must be linked with a MMMS library appropriate for the particular system. Libraries currently exist for IBM PC series and compatibles (requires the SERINT serial port handler, documented elsewhere), 4.2 and 4.3 BSD Unix and VAX/VMS.

LIST OF FUNCTIONS

MMMS exchanges data in *messages*. A message is composed of up to 1024 bytes of data – see *MS_SendMessage* or *MS_ReceiveMessage*.

Available functions are:

Name	*Description*
MS_Abort	Abandon any message being sent or queued
MS_Closedown	Shut down MMMS
MS_GetConfig	Read current protocol settings
MS_Init	Initialise library
MS_LastError	Return current error code
MS_ReceiveMessage	Read incoming message, if any
MS_SetConfig	Change current protocol settings
MS_SendMessage	Queue a message for sending
MS_Start	Try to establish communication
MS_Status	Return selected information on link status

Other manual pages:

Intro	This page

BUGS AND PROBLEMS

Reliability. The error detection used by the MMMS protocol is not as powerful as it could be, or should be. It seems to be perfectly adequate for use over directly-connected terminal lines.

The MMMS protocol isn't really suitable for use over split-rate modems, e.g. V.23 1200/75.

The library buffers messages received and messages to be sent internally. Buffer allocation is not terribly intelligent; in particular, a flood of short incoming messages can hog all the buffers and make it difficult to send messages.

The contents of incoming messages are not made available until the entire message has arrived. A full-sized message takes at least 30 seconds to arrive at 300 bps; while not a problem when, say, transferring files, the delay can be important if, say, the message contains a sequence of commands, the first of which could be executed while the others are coming in.

NAME

MS_Abort – abandon any message being or to be sent

SYNOPSIS

#include "mmms.h"

void MS_Abort()

DESCRIPTION

If any message is currently being sent, *MS_Abort* cancels it. Any messages queued for sending are also abandoned.

RETURN VALUE

None.

SEE ALSO

MS_SendMessage(MMMS)

NAME

MS_CloseDown – shut down MMMS interface and link

SYNOPSIS

#include "mmms.h"

void MS_CloseDown()

DESCRIPTION

MS_CloseDown() shuts down the MMMS interface and any established link, freeing all buffers and returning the serial port to its original setting. *MS_CloseDown()* must **always** be called before exiting a program using the MMMS system.

NOTES

It is safe to call *MS_CloseDown()* withtout first calling *MS_Init()*, so it is recommended to always call it before exiting, whether or not the interface was initialised. The effects of exiting without calling *MS_CloseDown()* are system-dependent; it will cause a system crash on the IBM PC implementation.

RETURN VALUE

None.

SEE ALSO

MS_Init(MMMS)

NAME

MS_GetConfig, MS_SetConfig – get and set various protocol settings

SYNOPSIS

#include "mmms.h"

struct MS_ConfigInfo MS_GetConfig()

void MS_SetConfig(config)

struct MS_ConfigInfo *config;

DESCRIPTION

MS_GetConfig returns a structure containing various protocol settings; *MS_SetConfig* allows those settings to be modified. *MS_GetConfig* and *MS_SetConfig* should be used to modify protocol settings after *MS_Init* has been called, but before the protocol is started with *MS_Start*. Note that modifying settings once communication has been established may cause the connection to be lost or data received to be corrupted.

NOTES

struct MS_ConfigInfo is defined in *mmms.h* as follows

```
struct MS_ConfigInfo
{
        char    soh;
        short   maxl, time, npad;
        char    padc, eolc, qctl, qbin, qrep;
        short   maxr, tim2;
};
```

The fields are as below. During initialisation they are exchanged, and the values used to control transmission; hence the values given are actually used by the other end of the link.

soh The control character indicating the start of a packet.

maxl The maximum number of data characters in a packet (up to 94).

time Time to wait before retransmitting a packet, in units of *ticks*, where there are 5 ticks in one second.

npad The number of padding characters to precede each packet sent.

padc The padding character to be used.

eolc An end of line character to be sent after each packet. If this is set to zero, no end of line character is required.

qctl Printable character to use to quote control characters. Briefly, control characters are sent in a printable representation, preceded by this quote character.

qbin Printable character to use to quote bytes with the most significant bit set. Some transmission lines may use this bit as a parity bit, so affected bytes are sent preceded by this character. This must not be the same character as *ctl* or *qrep*.

qrep If a succession of bytes in a message have the same value, they may be sent by sending the value preceded by a repeat count. This printable character introduces a repeat count. It must be a different character to *qctl* and *qbin*.

maxr Maximum number of times to reset the link before declaring it failed.

tim2 The maximum length of time to go without sending a packet to keep the link 'ticking over'.

It should rarely be necessary to alter any of the above from their default settings, and such alteration should not be made without reference to a description of workings of the protocol.

RETURN VALUE

MS_GetConfig returns the current configuration. *MS_SetConfig* has no return value.

SEE ALSO

MS_Init(MMMS)

NAME

MS_Init – initialise the MMMS library

SYNOPSIS

#include "mmms.h"

int MS_Init(debug_level)

int debug_level;

int MS_Start()

DESCRIPTION

MS_Init initialises the MMMS interface. Buffer space is claimed, the serial port is opened and set to the correct mode (if applicable) and the timers are started. *MS_Init* must be called before any other interface function. If *debug_level* is non-zero, it **may** cause debugging information to be recorded in a file *mmms.log*. This will not often be possible, but is implemented in the Unix MMMS library. Two levels of debugging information are currently available:

1 Record message traffic and any timeouts.

2 Also record packet traffic and protocol settings.

MS_Start initiates the protocol by causing a reset sequence to be started if the link is not yet active. After calling *MS_Start*, *MS_Status*(MMMS) should be called repeatedly to check the state of the link until either the link is established or the attempt deemed to have failed.

NOTES

The usual sequence to be followed when attempting to establish a link is to call *MS_Init*, then *MS_Start*, and then loop on *MS_Status* until either the link is established or an error occurs. The functions are split in this fashion to allow the application control while communication is being established; if a failure does occur, it may not be signalled until the appropriate timeout has expired, which could be a significant period of time.

To reiterate – MS_Init does **not** establish communication.

RETURN VALUE

MS_Init and *MS_Start* return 1 if successful, otherwise 0, in which case an error code may be obtained from *MS_LastError*.

SEE ALSO

MS_Status(MMMS), *MS_CloseDown*(MMMS)

ERRORS

MS_Init and *MS_Start* will fail for any of the following reasons:

MSF_NO_BUF_SPACE No space could be found for MMMS internal buffers.

MSF_LINE_SILENT Link timed out before connection established.

MSF_NO_TERMINAL The serial port could not be accessed.

MSF_TERM_PARAMS_ERR

The serial port could not be set for MMMS operation.

MSF_TIMER_NOT_SETUP

Unable to start timeout clock.

NAME

MS_LastError – return code of the last error

SYNOPSIS

#include "mmms.h"

char MS_LastError()

DESCRIPTION

MS_LastError returns a code for the last error to have occured. Errors may be either fatal error or warning errors (see NOTES below). The call resets the current error to NO_ERROR.

NOTES

Errors are defined in *mmms.h* and either fatal, necessitating the shutting down of the link, or warnings, which are recoverable.

Fatal errors are:

MSF_ADDQ_FULL, MSF_NO_MESSAGE

Internal queue errors.

MSF_BAD_MESSAGE_NO

A message with a bad sequence number has been received.

MSF_LINE_SILENT Too long has passed without a packet being received. The connection is assumed to be broken.

MSF_NO_BUF_SPACE No space available for internal buffers.

MSF_NO_TERMINAL Serial port could not be accessed.

MSF_TERM_PARAMS_ERR

Serial port could not be set for MMMS operation.

MSF_TIMER_NOT_SETUP

Unable to start timeout clock.

MSF_TOO_MANY_RESETS

Limit on number of resets exceeded.

The warnings that may be returned are:

NO_ERROR	No error has occured.
PROTOCOL_ERROR	Other side issued a reset
BAD_ACK	Non-existant packet acknowledged. Reset issued.
TOO_MANY_RETRIES	Packet resend limit exceeded, reset issued.

RETURN VALUE

One of the above error codes.

NAME

MS_ReceiveMessage – remove first message from receive queue

SYNOPSIS

#include "mmms.h"

int MS_ReceiveMessage(msg)

struct MS_Message *msg;

DESCRIPTION

MS_ReceiveMessage copies the message at the top of the queue of messages received (i.e. the oldest message received) to *msg*.

NOTES

struct MS_Message is defined in *mmms.h* as follows:

```
struct MS_Message
{
        int     length;
        char    data [MAX_MESS_LEN];
};
```

length holds the number of bytes of data in the message. The message itself is placed in *data*. The maximum length of a message is 1024 bytes.

RETURN VALUE

MS_ReceiveMessage returns 0 if successful, 1 if no message has arrived and -1 if a fatal error has occured, in which case an error code may be obtained from *MS_LastError*.

SEE ALSO

MS_SendMessage(MMMS)

ERRORS

MS_ReceiveMessage will report a fatal error for any of the following reasons:

MSF_ADDQ_FULL, MSF_NO_MESSAGE

Internal queue error.

MSF_BAD_MESSAGE_NO

Received message with bad sequence number.

MSF_LINE_SILENT Link timed out before connection established.

MSF_TOO_MANY_RESETS

Limit of line resets exceeded.

NAME

MS_SendMessage – add a message to the queue to be sent

SYNOPSIS

#include "mmms.h"

int MS_SendMessage(msg)

struct MS_Message *msg;

DESCRIPTION

MS_SendMessage copies the message pointed to by *msg* to an internal buffer and places it in the queue of messages to be sent.

NOTES

struct MS_Message is defined in *mmms.h* as follows:

```
struct MS_Message
{
        int     length;
        char    data [MAX_MESS_LEN];
};
```

length holds the number of bytes of data in the message. The message itself is placed in *data*. The maximum length of a message is 1024 bytes.

RETURN VALUE

MS_SendMessage returns 0 if successful, 1 if there is no buffer space currently available and -1 if a fatal error has occured, in which case an error code may be obtained from *MS_LastError*.

SEE ALSO

MS_ReceiveMessage(MMMS)

ERRORS

MS_SendMessage will report a fatal error for any of the following reasons:

MSF_ADDQ_FULL, MSF_NO_MESSAGE

Internal queue error.

MSF_BAD_MESSAGE_NO

Received message with bad sequence number.

MSF_LINE_SILENT Link timed out before connection established.

MSF_TOO_MANY_RESETS

Limit of line resets exceeded.

NAME

MS_Status – return information on current link status

SYNOPSIS

#include "mmms.h"

struct MS_StatusInfo MS_Status()

DESCRIPTION

MS_Status returns a structure (see NOTES) containing selected information on the current status of the link.

NOTES

struct MS_Status is defined in *mmms.h* as follows:

```
struct MS_Status
{
        short   resets;
        short   ms_sent, ms_rec;
        short   pk_sent, pk_rec, pk_bad;
        short   NAK_sent, NAK_rec;
        char    link_state;
};
```

resets The number of times the link has been reset.

ms_sent The number of messages sent.

ms_rec The number of messages received.

pk_sent The number of packets sent.

pk_rec The number of packets received without error.

pk_bad The number of corrupted packet received, i.e. packets with incorrect checksum values.

NAK_sent The number of Negative ACknowledgements, or requests for retransmission, sent.

NAK_rec The number of NAKs received.

link_state The current state of the link.

The possible link states are:

LS_IDLE Link is working but no traffic is present.

LS_SEND A message is being sent.

LS_REC A message is being received.

LS_SENDREC

Messages are being sent and received.

LS_RESET The link is being reset.

LS_DOWN The link has failed, or has not been started.

LS_FATAL A fatal protocol error has occured. See *MS_LastError*.

RETURN VALUE

Current link status.

SEE ALSO

MS_LastError(MMMS)

Appendix E

FlAP Packet Details

This appendix gives details of FlAP packet types and parameter data. The general format of FlAP packet interaction is given in §3.3.1; this appendix confines itself to details of packet contents. The packet type for each function is given as the function ID. The returned data for each function is in addition to the result and error code always returned.

Each item of parameter or returned data is listed with its data type. Entries with a data type of byte(s) indicate the item contains zero or more bytes — the actual number number is determined by the value of a previous field.

Further explanation on some functions, indicated by a • after the function name, is given following the table immediately below.

Function	ID	Parameter data		Returned data
flqAbort	'A'			
flqAddRec	'B'	fd	byte	
		buffer size	short	
		buffer contents	byte(s)	
flqVAdd•	'C'	fd	byte	
		no. records	short	
		rec offset	long	
		rec whence	byte	
		rec length	short	
		rec contents	byte(s)	

Table E.1: Continued ...

Function	ID	Parameter data		Returned data
fiqClose	'D'	fd	byte	
fiqCompare	'E'	from fd	byte	
		to fd	byte	
		no. records	long	
		match	byte	
fiqCopy	'F'	from fd	byte	
		to fd	byte	
		no. records	long	
		forwards	byte	
fiqCreate•	'G'	filename len	byte	
		filename	byte(s)	
		filetype	byte	
		record length	short	
		no. keys	byte	
		key flags	short	
		key no. segs	byte	
		key seg start	short	
		key seg length	short	
fiqDelFile	'H'	filename len	byte	
		filename	byte(s)	

Table E.1: Continued ...

Function	ID	Parameter data		Returned data	
flqDelRec	'I'	fd	byte		
		no. records	long		
flqDuplicate	'J'	fd	byte		
		no. records	long		
		to offset	long		
flqFindRec	'K'	fd	byte	record length	short
		index no	byte	record contents	byte(s)
		match type	byte		
		key offset	short		
		key size	short		
		key contents	byte(s)		
		buffer size	short		
flqGetRec•	'L'	fd	byte	record length	short
		buffer size	short	record contents	byte(s)
flqVGet•	'M'	fd	byte	record length	short
		no. records	short	record contents	byte(s)
		rec offset	long		
		rec whence	byte		
		rec buffer size	short		

Table E.1: Continued ...

Function	ID	Parameter data		Returned data	
flqOpen	'N'	filename len	byte		
		filename	byte(s)		
		filetype	byte		
		record length	short		
flqRenFile	'O'	from fname len	byte		
		from filename	byte(s)		
		to fname len	byte		
		to filename	byte(s)		
flqSearch•	'P'	fd	byte	no. bytes	short
		regex length	short	record contents	byte(s)
		regex	byte(s)		
		no. recs	long		
		forwards	byte		
		buffer size	short		
flqSeek	'Q'	fd	byte		
		offset	long		
		whence	byte		
flqTruncate	'R'	fd	byte		

Table E.1: Continued ...

Function	ID	Parameter data		Returned data
flqUpdateRec	'S'	fd	byte	
		buffer size	short	
		buffer contents	byte(s)	
flqVUpdate•	'T'	fd	byte	
		no. records	short	
		rec offset	long	
		rec whence	byte	
		rec length	short	
		rec contents	byte(s)	

Table E.1: FlAP packet types and contents

flqVAdd, flqVUpdate The record offset, whence, length and contents sequence is repeated for each record involved.

flqCreate The key segment start and length sequence is repeated for each key segment. Key flags and segment details are repeated for each key.

flqGetRec The number of record contents bytes returned is the minimum of the record length and the buffer size.

flqVGet The record offset, whence and buffer size sequence is repeated for each record requested. The returned record length and contents sequence is also repeated as required.

flqSearch The length of the record found is not returned, just the record contents up to a maximum of *buffer size* bytes.

Appendix F

FlAP Application Writer's Guide

1. What is FlAP?

FlAP is a file access system which allows microcomputer-based applications to access mainframe files. It consists of an interface library on the microcomputer connected to a file server on the mainframe by a normal terminal line. The current implementation allows IBM PC or compatible microcomputers to access files held on Unix systems. The aim of this guide is to acquaint the PC application writer with FlAP's capabilities and how it may be used in an application. It concludes with instructions on running FlAP applications using the current PC-Unix implementation, and should be read in conjunction with the FlAP Programmer's Manual, which provides manual pages for each available FlAP function and other facets of the FlAP system.

FlAP is designed for use over terminal lines. The speed at which data may be transferred over these links is hundreds or thousands of times slower than for normal disc accesses, so to remain effective FlAP departs from more conventional microcomputer disc access systems in three main ways.

1. File data is accessed in units of records. Rather than treat the file as merely a sequence of bytes (i.e. a *bytestream*), FlAP considers files to be composed of a sequence of records. Records may be of a fixed length or vary in length throughout the file. When dealing with variable-length records, e.g. lines of text, this allows the application to restrict data transfers across the link to just the data required, e.g. ask for line 1 rather than a number of bytes that may be either too few, necessitating a fresh request, or too many, wasting the link bandwidth.

2. The provision of *extended file operations*. Some common file operations, usually performed using a sequence of conventional file manipulation functions, may require a lot of data to be read from or written to the file(s). For example, searching a file is done by reading file data until the sought-for sequence is found. If a 20k file is being searched for a word that is not, in fact, in the file then even with a 960 cps† line (one of the fastest speeds in use) it would take 22 seconds to conclude the search

† cps = characters per second.

had failed. If the link is via a modem over a dial-up line, the line speeds are more likely to be 120 or 30 cps in which case the times taken would be nearly 3 minutes and over 11 minutes respectively. FIAP extended operations, which include searching, are done in the file server and so avoid having to transmit data and the consequent delays.

3. FIAP file functions are *asynchronous*. In a normal file system, the available file functions are synchronous; that is, the application requests, say, data be read from a file by calling a function that does not return until the data has been read. The application loses control of the system while the function is in progress. This is not serious when file operations are usually finished in at most a second or two. When they could take 10, 20, 30 or more seconds, then the application may well be able to get on with other activities while waiting, including monitoring for a signal from the user to abandon the request for data.

2. FIAP file and record types

There are two components determining the type of a FIAP file, the record type and the file organisation. There are four record types.

1. *Fixed-length*. Records occupy a fixed number of bytes. The record length is specified when the file is created.
2. *Variable-length 1*. Records occupy a varying number of bytes, with the length of each record being stored at the start of the record.
3. *Variable-length 2*. An alternate format of variable-length records, useful for reading and writing ANSI standard tape files.
4. *Text*. A variable-length record format where one record corresponds to one line of text. The file server removes end-of-line indicators on reading and replaces them on writing.

These may be stored in one of three file organisations.

1. *Sequential*. Records are stored in sequence one after the other. To access a particular record it may be necessary to read through all the previous records.

2. *Relative.* Records are stored in sequence, but in such a way that a particular record may be accessed directly. For example, variable-length records may be held in cells of a fixed size.

3. *Indexed.* A portion of the record contents determine the record *key*. The key is stored in one or more indexes, which are structured in such as way that keys may be rapidly located within the index. Records may be retrieved by specifying their key value.

Not all record types may be available with a particular file organisation. FlAP in fact offers 9 file types, named as indicated in the following table.

	File organisation		
Record type	Sequential	Relative	Indexed
Fixed-length	FL_SEQ	FL_REL	FL_IND
Variable-length 1	VL1_SEQ	VL1_REL	VI1_IND
Variable-length 2	VL2_SEQ	—	—
Text	TXT_SEQ	TXT_REL	—

Table F.1: FlAP record types

Text files deserve a few more words of explanation. They exist as a separate category because of the widely varying number of ways in which mainframe filing systems store text. By putting them into a separate category, the file server can take care of such representational matters, allowing application programs using text to be used with file servers on different mainframes. A sequential text file is thus a normal readable text file. Relative text files are intended to be used merely as temporary work files while text is being processed. They allow lines to be deleted and new ones to be inserted as well as to be accessed directly, and should really be regarded as indexed files, the indexing being done on the line number (not that line numbers are ever explicitly assigned anywhere).

The FlAP Programmer's Manual also mentions STREAM files. These are still experimental; they are intended to allow unformatted access to a file, and their final form is not yet fixed.

3. Using FlAP functions

The only current interface library is written in C, and intended for use by C programs. These notes assume that the application is to be written in C, and so all illustrations are given in that language.

Before any FlAP functions are called, the program must include the FlAP header file, *flap.h*. This is most conveniently stored in the normal *include* directory; a program using FlAP might begin like this.

```
/*
 *      A FlAP application
 *
 *      Created:        99/99/99
 *
 */

#include <stdio.h>
#include <flap.h>
```

Figure F.1: The start of a FlAP program

The header file declares the various structures used by FlAP functions, and defines various constants. If the preprocessor symbol *__STDC__* is defined, ANSI C function prototypes are given for all FlAP functions; if not, normal function declarations are given.

3.1. Starting up

Before any FlAP functions can be used, contact with the file server must be established. This is done using three functions. *flInit* initialises the interface library and the communication link. *flStart* then begins the attempt at establishing communication with the file server. Finally, *flReady* reports on the progress of the attempt, whether it has succeeded, or is still in progress, or has failed.

When the application has finished, it *must* call *flShutdown* to close down the communication link and the server. If the application exits without shutting down, the current PC implementation will crash the machine (it must be given the chance to remove various interrupt vectors used for timing). So a minimal FlAP program that just initialises and then immediately shuts down the server and exits could be written like this (building on Figure F.1):

```
/* Fatal. A fatal error has taken place. Close down and exit,
   printing an error message and returning an exit status of 1 to
   indicate failure. */
void Fatal (message)
char *message ;
{
   flShutdown () ;
   fprintf (stderr, "Fatal error: %s\n", message) ;
   exit (1) ;
}

/* StartFlap. Try to establish contact with the FlAP server.
   Initialise the FlAP library, start attempts at communication
   with the server and wait for either success or failure. */
void StartFlap ()
{
   if ( flInit () < 0 )
      Fatal ("flInit failed") ;
   if ( flStart () < 0 )
      Fatal ("flStart failed") ;

   while ( flReady () == 0 )
      ;

   if ( flReady () < 0 )
      Fatal ("Cannot establish contact with server") ;
}


/* Main. Establish contact and shut down again, returning an exit
   status of 0 to indicate success. */
main ()
{
   StartFlap () ;
   flShutdown () ;
   printf ("Successful !\n") ;
   exit (0) ;
}
```

Figure F.2: A minimal FlAP program

3.2. Basic file and record operations

FlAP's view of a file is of a numbered sequence of records, the first record being record 0. FlAP maintains a notion of *currency*, an imaginary pointer into the file indicating the record that will be accessed by the next read or write. This is analogous to the Unix/MS-DOS bytestream file model, with the unit of access being the record rather than the byte.

The exact nature of currency, and the extent to which the currency can be manipulated, depends on the type of the file in question. For example, currency is usually a reflection of the physical position of a record within the file. However, in the case of indexed or relative text files, currency reflects the position of the record key in the index.

In some cases it maybe impossible to obtain the currency in terms of a record number. For example, it is possible to skip straight to the end of a variable-length record file, but not possible to know how many records have been passed over. In these circumstance, currency still exists, but is assigned a numeric value of DONTKNOW.

The file type also affects which operations are available and which it is sensible to use. For example, searching a file index for a record key only makes sense if the file is an indexed file, and moving currency back one record in a sequential file may involve rewinding the file (setting currency to the first record) and reading records until the desired record is located.

flAddRec	Add a record to a file
flClose	Close an open file
flCreate	Create a new file
flDelRec	Delete one or more records
flFindRec	Retrieve a record by its key
flGetRec	Read a record
flOpen	Open an existing file
flSeek	Reset file currency
flUpdate	Rewrite a record

Table F.2: FlAP basic operations

The basic FlAP operations resemble those of Unix or MS-DOS, and are listed in Table F.2 above. Indexed files which permit duplicate keys, and relative text files, need to distinguish between adding new records and rewriting old ones, hence the need for two functions. Deletion of records is only appropriate for

indexed and relative text files. For other file types, addition of a record just means moving to the end of the file and writing the record, while attempting to delete records generates an error. Similarly, retrieving a record by its key value makes no sense for files that are not indexed, and so generates an error for other file types.

All FlAP operations (with the exception of *flShutdown*) return an status value. If an error has occured, this will be negative, and further information on which error has occured may be found in the global variable *flErrno*, which is declared in *flap.h* along with FlAP error codes. An example code fragment:

```
if ( flAddRec (fd, reclen, recbuf) < 0 )
   switch ( flErrno )
   {
      case EIDXSPC
      case ECNTXTND : Error ("File system full") ;
                      break ;

      case ENODUPS  : Error ("Duplicate key") ;
                      break ;

      default       : Fatal ("flAddRec failed") ;
                      break ;
   }
```

Figure F.3: Inspecting FlAP error codes

3.3. Extended and vector operations

As mentioned above, FlAP provides several *extended operations*, where the file server performs operations that would otherwise involve timeconsuming data transfers. There are currently four extended operations.

flCompare	Compare records in two files
flCopy	Copy records from one file to another
flDuplicate	Duplicate records in a file
flSearch	Search records for a pattern

Table F.3: FlAP extended operations

It is also frequently necessary to read and write multiple records at once — for example, reading a screenful of text in an editor. If this has to be done by issuing multiple read or write requests (perhaps each preceded by a seek), it may be more convenient to combine these into one request. This will also be faster, as it avoids the delays between finishing one read or write and starting the next due to the time taken for the result to be returned and the new request to reach the file server. To this end, FlAP provides three *vector operations* which perform multiple read and writes in one operation, controlled by a vector of requests. The vector operations are:

flVAdd	Add multiple records
flVGet	Read multiple records
flVUpdate	Rewrite multiple records

Table F.4: FlAP vector operations

These are equivalent to a series of seek and read/write requests. If an error occurs, then processing stops and the result returned indicates the record being processed at the time.

3.4. Asynchronous vs. Synchronous operations

To give an idea of how long a typical FlAP function will take to execute, the following table gives times for sample functions at differing line speeds. The line speed is in characters per second and all times are in seconds. The file was a fixed-length relative file (FL_REL) with a record length of 200 bytes. The *flVGet* function was used to read 10 records at once — the time given is the time taken per record i.e. the time taken by the call divided by the number of records read.

Comparison of the times for *flGetRec* and *flVGet* illustrates the improvement in throughput obtainable using vector operations. Remember that a vector read on one record is equivalent to a seek followed by a read.

Speed	flCreate	flClose	flOpen	flSeek	flAddRec	flGetRec	flVGet
960	0.7	0.5	0.5	0.5	0.8	0.9	0.4
480	0.7	0.5	0.5	0.5	0.9	1.1	0.7
240	0.8	0.5	0.7	0.5	1.5	1.6	1.1
120	0.9	0.7	0.7	0.7	2.7	2.7	2.1
30	1.8	1.2	1.6	1.3	9.2	9.3	8.2

Table F.5: FlAP function timings

However, it must be stressed that the times given above were obtained under near-ideal conditions. If the connection is noisy the times will increase considerably. If conditions deteriorate to the point where the connection is lost, minutes may elapse before the interface finally gives up on the connection and reports an error.

To enable the application to remain in control, and perhaps carry out useful work while the FlAP operation progresses, all FlAP file and record operations are available using asynchronous functions. Calling one of these functions (they are named *flq?????* rather than *fl?????*) initiates the requested operation and returns control to the application straight away. The application must then call *flDone* to check on the progress of the operation. The current implementation also relies upon *flDone* to process incoming and outgoing data, so it must be called reasonably often. Only one FlAP operation may be in progress at any one time — the interface library does not queue requests.

There are three asynchronous support functions in all.

flqAbort	Request current operation be aborted
flDone	Check on progress of current operation
flWait	Wait for current operation to complete

Table F.6: FlAP asynchronous support functions

The code fragment below illustrates how asynchronous functions can be used to avoid the application

losing control of the user interface.

```
/* Open the workfile and monitor the keyboard for a user    *
 * interrupt with the function UserInt, which returns TRUE *
 * if one occurs.If this happens, exit via Fatal.           */

if ( flqOpen (WORK_NAME, WORK_TYPE, WORK_RECLEN) < 0 )
   Fatal ("flqOpen failed") ;

while ( ( done = flDone (&fd) ) > 0 )
   if ( UserInt () )
      Fatal ("Interrupted") ;

if ( done < 0 )
   Fatal ("flqOpen failed") ;
```

Figure F.4: A use of FlAP asynchronous functions

Asynchronous functions can also be used to reduce the time the user must wait for the file system. The following example illustrates one possible approach.

The FlAP screen editor, RBed, uses synchronous functions to open its files and to read text. However, it uses asynchronous functions to rewrite/add/delete individual lines. Changes to the current line are monitored, and if the line has been altered then when attention moves to another line it is rewritten, added or deleted as necessary. This is done by first waiting for any asynchronous operation in progress to complete, checking the returned result and error code against those predicted, and then requesting an asynchronous rewrite/add/delete. In practice, the vast majority of such asynchronous operations complete while the user is editing the next line of interest, and so the file access does not delay editing, even at line speeds of 30 cps†.

RBed only allows one asynchronous operation to be outstanding at once. If two lines — perhaps blank lines — are added in quick succession, this can give rise to brief delays. A little modification to RBed would allow more than one request to be outstanding at once, reducing delays still further. Another possibility would be to read a few possibly useful lines not currently held in the PC whenever no other FlAP operation is active, hopefully reducing the number of delays while reading lines in.

† Of course, if once the line is edited the user jumps to a line not currently held in the PC, a delay ensues while the asynchronous operation completes and the lines are read in (lines are always read in groups). Asynchronous operations cannot help if nothing further can be done until the operation completes.

To summarise, carrying out FlAP operations asynchronously can help reduce delays to the user by carrying on file access in parallel with other activities. It may not always be possible, and if possible may not be easy, but can pay dividends.

4. Application startup

The PC interface library uses the SERINT serial port interface (described elsewhere). If used with the VT100 terminal emulator that accompanies SERINT, FlAP applications may be started from the Unix command line. A special protocol is used to signal a request for a FlAP application to the PC and pass the application name and arguments. Alternatively, an application may be started manually.

4.1. Manual startup

To start a FlAP application manually, the Unix FlAP server *flap* must be started first. *flap* takes one argument *-ddddd* requesting debugging information be generated — the number of *d*s indicating the level of debugging infomation required. The available levels of debugging range from *-d*, which logs requested FlAP function names to a logging file *flap.log*, to *-ddddd* which also logs FlAP function parameters and packet contents, and logs MMMS† packet traffic to *mmms.log*. Any other arguments are ignored.

Once the server has been started, the terminal emulator must be exited and the FlAP application started from the MS-DOS command line in the normal way. Since the FlAP interface uses SERINT, that must have been already loaded and in command of the serial port. Use of terminal emulators other than any using SERINT may cause trouble here if they do not leave the serial port exactly as they found it. It is recommended that the terminal emulator accompanying SERINT is used, which in turn allows automatic application startup.

When the application terminates, *flShutdown* instructs the server to terminate. Before doing so, the server waits for a couple of seconds to allow a terminal emulator time to regain control.

† MMMS is a protocol used by FlAP to provide error-free communication over the terminal line.

4.2. Automatic startup

Several components must be used to allow FlAP applications to be started from the Unix command line.

The terminal program must have been invoked by the PC driver program TERMINAL. This runs the terminal emulator VT100 and also handles the loading and execution of FlAP application programs, which must be located either in the current directory or in one of the directories specified by the MS-DOS PATH environment variable.

A copy of the Unix program *start* must be renamed to the name of the application. If another FlAP application is present, it is simpler to create a link to its copy of *start* with the new name using the Unix command *ln*. When run, *start* exchanges the application name (i.e the name *start* was actually called under) and any arguments with TERMINAL. It then invokes *flap*, also passing on any arguments. If the terminal emulator program being used does not support automatic startup of FlAP applications, the message 'FlAP: FlAP operations not supported. Press RETURN.' appears instead and *start* exits.

5. Compiling FlAP applications

The FlAP interface functions for the PC are held in a library, FLAP.LIB. They in turn require the MMMS functions held in MMMS.LIB. So when compiling a FlAP application both libraries will need to be included at the linking stage.

The libraries are both compiled using the Zortech C v2.0, small model†. Assuming the same compiler is used for the application, the Zortech compiler control program *zc* can be used to compile and link the application thus (this assumes that both libraries are held in the library directory indicated by the PC-DOC environment variable LIB):

```
C>zc prog.c flap.lib mmms.lib
```

Figure F.5: Compiling and linking a FlAP application

† For the benefit of newcomers to the baroque delights of the Intel iAPX 86 family, the different programming models reflect tradeoffs between code simplicity and the need for over 64k of either code or data or both.

Larger applications may need to use other memory models. It is important to remember that the FlAP and MMMS libaries used with such applications must themselves have been compiled using the relevant memory model. Following normal Zortech C library naming conventions, the libraries are named FLAPP.LIB and MMMSP.LIB (program model), FLAPD.LIB and MMMSD.LIB (data model) and FLAPL.LIB and MMMSL.LIB (large model).

Appendix G

Current dial-up communication speeds

1. Transmission mechanisms

Standards for modems for use over the public dial-up telephone network are set by the CCITT. The following table summarises the major adopted recommendations — details may be found in [Jennings86].

V.21	300 bps full-duplex
V.23	1200 bps half-duplex with 75 bps reverse channel
V.22	1200 bps full-duplex
V.22*bis*	2400 bps full-duplex
V.32	9600 bps full-duplex

Table G.1: CCITT dial-up modem standards

At the time of writing, V.21 and V.23 predominate in the UK. However, prices of V.22 and V.22*bis* modems have been falling, and are beginning to approach the level of the slower modems.

V.32, on the other hand, requires sophisticated echo cancellation facilities necessitating a considerable development effort. It is only recently that chipsets implementing the V.32 standard have become available to modem manufacturers; currently available V.32 modems use proprietary technology, and are consequently expensive.

To meet the demand for high-speed modems, some manufacturers have produced half-duplex 9600 bps modems either based on V.32 or V.29, a standard for full-duplex 9600 bps operation on 4-wire leased lines. The latter, in its half-duplex two-wire guise, has lately come into widespread use as the transmission standard used in facsimile machines, and consequently chipsets are widely available and cheap. To enable full-duplex use, the modem regularly reverses the channel, either at fixed intervals (ping-pong) or by statistical duplexing — the modems communicate via a 300 baud reverse channel and agree to reverse the main channel after exchanging information on the amount of data waiting in their buffers).

Others have produced multicarrier or Packetised Ensemble Protocol (PEP) designs. These also operate in half-duplex, but rather than use one carrier, they use 60 or more, each modulated slowly. Data is sent in parallel along these channels under protocol control. Whereas on a more conventional modem a noisy line may force it to fall back to a less sensitive standard — say from 9600 bps operation to 7200 bps or 4800 bps — PEP modems can abandon channels one at a time, and so (it is claimed) come closer to maintaining maximum possible data throughput. It is also claimed that this makes them less likely to have to abandon connections altogether†.

2. Error correction

As transmission speed increases, the susceptibility of the data to corruption by bursts of line noise increases. To counter this, higher speed modems increasingly use some form of error detection and correction protocol to present the user with an error-free link. The most common protocol currently in use is Microcom Inc.'s MNP protocol suite, which offers increased capability in ascending classes (see Table G.2 overleaf).

Class 3 modems and above send data synchronously rather than asynchronously, giving around a 20% increase in throughput. Adaptive packet assembly (Class 4) allows protocol packet sizes to be dynamically adjusted to suit prevailing line conditions. Class 6 allows modems to select the highest mutually acceptable speed, and, if necessary, perform statistical duplexing. The various currently-defined MNP classes are listed in Table G.2 overleaf.

The CCITT is currently active in this area, and is due to ratify formally recommendation V.42 in November 1988. This is an error detection and correction protocol for use between modems, named LAPM (Link Access Protocol — Modem). It is a variant of the LAPB HDLC protocol used in X.25 level 2. If, during the initial handshake, it turns out that one modem does not support V.42, the standard requires the conforming modem to fall back to MNP.

† One user reports that in an *ad hoc* test, his PEP modem could only be forced to drop the connection by continous tone dialling on an extension accompanied by sustained shouting.

Class	Description
1	Asynchronous half-duplex
2	Asynchronous full-duplex
3	Synchronous full-duplex
4	Adaptive packet assembly
5	Data compression
6	Higher-speed negotiation

Table G.2: MNP Classes

3. Throughput enhancement

Many error correcting modems also employ data compression in one form or other to further increase throughput. Some impressive claims have been made in this regard with several manufacturers reporting sustained quadrupling of throughput†.

Several conventional asynchronous protocols, notably those which require regular acknowledgements to short packets, do not work well over modems that fake a full-duplex link over a half-duplex link, because of the frequent delays incurred while the channel is reversed. Some such modems address the problem by providing support for specific protocols, relying on the modem's own error correction protocol to ensure error-free transfers. The effect of such support can be dramatic — one PEP modem is reported to attain throughputs of over 12,000 bps on an international dial-up line using this 'protocol spoofing', while throughput without was less than a tenth of this.

4. The future

A recent comparison of high-speed dial-up modems [Humphrey88] predicts that V.32 modems will become increasingly popular as, based on experience with V.22 and V.22*bis* modems, the availability of chipsets makes design of V.32 modems easier and production cheaper. Allied to the data compression

† Telcor Inc. claim 9600 bps throughput on a 2400 bps modem.

efficiencies currently being claimed, modems with 34,800 bps peak throughput are forecast for the near future. However, these will probably take some time to come into general use — as noted above, V.22 and V.22*bis* modems have still to replace V.23 and V.21 in the UK, so widespread use of V.32 modems can hardly be said to be just around the corner.

Further ahead, the Integrated Services Digital Network (ISDN) [Stallings85] promises data transfer rates starting at 64 kbps. However, it will doubtless be some considerable time before this replaces the dial-up network to the point where conventional voice-band modems are no longer required.

Appendix H

ASCII Table

Dec	Hex	Control	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	@	NUL	32	20	SPC	64	40	@	96	60	'
1	01	A	SOH	33	21	!	65	41	A	97	61	a
2	02	B	STX	34	22	"	66	42	B	98	62	b
3	03	C	ETX	35	23	#	67	43	C	99	63	c
4	04	D	EOT	36	24	$	68	44	D	100	64	d
5	05	E	ENQ	37	25	%	69	45	E	101	65	e
6	06	F	ACK	38	26	&	70	46	F	102	66	f
7	07	G	BEL	39	27	'	71	47	G	103	67	g
8	08	H	BS	40	28	(	72	48	H	104	68	h
9	09	I	HT	41	29	)	73	49	I	105	69	i
10	0A	J	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	K	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	L	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	M	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	N	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	O	SI	47	2F	/	79	4F	O	111	6F	o
16	10	P	DLE	48	30	0	80	50	P	112	70	p
17	11	Q	DC1 XON	49	31	1	81	51	Q	113	71	q
18	12	R	DC2	50	32	2	82	52	R	114	72	r
19	13	S	DC3 XOFF	51	33	3	83	53	S	115	73	s
20	14	T	DC4	52	34	4	84	54	T	116	74	t
21	15	U	NAK	53	35	5	85	55	U	117	75	u
22	16	V	SYN	54	36	6	86	56	V	118	76	v
23	17	W	ETB	55	37	7	87	57	W	119	77	w
24	18	X	CAN	56	38	8	88	58	X	120	78	x
25	19	Y	EM	57	39	9	89	59	Y	121	79	y
26	1A	Z	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	[	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	\	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	]	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	^	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	_	US	63	3F	?	95	5F	_	127	7F	DEL

Table H.1: ASCII Table