EVALUATION OF A
SCREEN MANAGEMENT
PROTOCOL

Supervisor: Dr M. A. Maclean.

T. J. Parish
October 1986
0. CONTENTS.

1. INTRODUCTION.

2. OBJECTIVES.

3. A DESCRIPTION OF SSMP.

4. THE IMPLEMENTATION.
   4.1 Implementation Decisions............ 4.1
   4.2 The Simulated Data channel........... 4.2
   4.3 The Terminal manager.................. 4.3
   4.4 The Host Application Interface....... 4.5

5. EVALUATION OF SSMP.
   5.1 The Users' view.......................... 5.1
   5.2 The Application Programmers' view... 5.4
   5.3 Keystroke Interpretation and Binding. 5.6
   5.4 Terminal Manager Functions............ 5.8
   5.5 Data Link Management................... 5.11
   5.6 Performance with Chef.................. 5.13
   5.7 Sumarising SSMP......................... 5.15

6. DESIGNING A BETTER PROTOCOL.
   6.1 What is required?....................... 6.1
   6.2 Custom Built Terminal Managers....... 6.3
   6.3 A Proposed alternative................ 6.4
   6.4 Implementing DSMP....................... 6.11

7. EVALUATION OF DSMP.
   7.1 An Overview.............................. 7.1
   7.2 Measurements with Chef.................. 7.3

8. SUMMARY.

9. APPENDICES.
   A References.
   B A Summary of Terminal Manager Functions.
   C Technical Details of DSMP.
   D DSMP Host Toolbox.
   E Results from SSMP and DSMP Experiments.
   F Examples of the Data encoding Schemes.
   G The Terminal Manager Program.
   H The Host Application Toolbox.
   I The Host Application Interface.
   J The SSMP Programs.
1. INTRODUCTION.

The development of large and complex computer networks and the increased popularity of full screen editors and other data manipulation utilities has created certain difficulties for users on simple remote terminals. Network delays and the cost of data transmission across these networks have made simple text editing and related operations very slow and expensive.

A number of methods can be used to reduce the cost of such editing operations, the simplest of which is to use only batch or line-mode editors instead of full screen devices. However this solution is unsatisfactory, since users have grown to expect screen editors to be available. Also, most methods used to increase the speed of screen editing operations across a network from a remote terminal will increase the cost of data transmission.

In recent times, computer users have come to expect an increasing degree of user friendliness. This expectation usually includes access to a responsive full screen editor for text editing and word processing as well as a fast friendly consistent form of data entry to specific applications. This interactive software is expected to have such features as windowing, cursor movement, and a rich set of image editing functions.

The Simple Screen Management Protocol (SSMP), is a protocol sponsored by the United Kingdom Joint Network Team (JNT) [SSMP85] to efficiently and transparently perform full screen management in a fashion compatible with packet switched networks. SSMP fits into the session and presentation layers of the ISO-OSI network architecture model and thus is not concerned with error detection, network routing or any aspect of the actual physical transportation of data.

The protocol specifies that processing occur at two physical locations in a network. Within a host computer, a network interface module allows the host applications software, editors, etc. to communicate with remote terminals, and, at a location logically close to the user's terminal, a terminal manager controls the functions of the terminal. The manager could operate within a concentrator for terminal clusters or within a micro computer as part of the terminal itself.

As well as providing a full description of the functions that are to be provided by the terminal manager and host application interface, SSMP specifies exactly the format of the data to be sent between the two processing components. The degree of specification here is to a very low level. This is done in an attempt to allow portability of applications programs between installations.

The SSMP definition was released in July 1985 [SSMP85] making it a very recent contribution to the network protocol landscape. Protocols sponsored by large committees produce a great deal of expectation from implementors and users alike. The objective of this project was to implement SSMP and evaluate its performance.
2.1 Objectives

The Simple Screen Management protocol (SSMP) defines a method by which a host application program wishing to perform full screen management of a remote terminal, can communicate with that terminal in an efficient manner. The protocol is fully described in the Fawn book [SSMP85] which is the official definition of the protocol and fully describes all of the functions of the protocol. Although this reference is called the Fawn book, SSMP is not one of the "coloured book protocols". This term refers to a series of protocols defining electronic mail standards, and other similar tasks. It is recommended that the reader of this report be familiar with the Fawn book but this is not a necessity.

The objective of this project was to implement an experimental version of the Simple Screen Management Protocol and to evaluate its performance. This evaluation considered both the networking and functional aspects of the protocol as well as aspects such as the users' and programmers' views of the protocol and the portability of applications that use it. Then, from this evaluation, a new protocol was designed that improved upon the faults found with SSMP, and provided a more efficient and flexible service to the host application programs and to the remote users. This protocol, the Distributed Screen Management Protocol (DSMP), was then implemented and evaluated in order to compare it to SSMP.

The University of Canterbury Computer Science department's VAX 11/750 computer was chosen to develop both the experimental SSMP and DSMP systems. All of the programs were written in the programming language 'C' under the UNIX™ operating system.

The implementation involved designing and building a terminal manager to control the physical terminal screen and user interface. This included a high level of "intelligence" to perform the functions required by SSMP. Following this, a simulated communications channel and a host application interface were created to allow host programs to communicate with the terminal manager. Finally, a host application program was modified to operate in a fashion consistent with SSMP. Because of the complex nature of real data networks, it was decided to simulate a data channel between the two components of the system within a single computer.

Almost all of the features specified by the SSMP definition [SSMP85] were implemented so that as many of the features as possible could be tested. A number of features, however, were deliberately left out either because their contribution to the overall practical functionality of the protocol is minimal, or because they were outside the scope of the evaluation which was not concerned with many minor aspects of the protocol. The evaluation of SSMP was mainly concerned with:

• User command response delays
• Terminal manager transparency
• Portability of host applications programs
• Programmability - The complexity of the task of interfacing with SSMP.
2.2 Objectives

- The flexibility of the protocol
- The functionality of the terminal manager
- Session control
- Data encoding and transmission efficiency

From this critical examination of the Simple Screen Management protocol, a proposed new protocol was designed. This protocol is based upon SSMP, but a number of its features will be very different. This new protocol is constructed as a practical and efficient alternative to SSMP. The features included in this new protocol will be more general and provide a much more flexible set of functions than those provided by SSMP. The proposed protocol is also implemented and evaluated in the same fashion as described above in an effort to compare it with SSMP.

As remote terminal editing becomes more popular, the necessity for a protocol to control these editing sessions grows. The question of whether screen management protocols such as SSMP will become the standard or whether other more complex protocols will succeed is addressed in this report. The question of whether any protocol at all is required between a host computer and a terminal is also discussed.
3. A DESCRIPTION OF SSMP.

The Simple Screen Management Protocol (SSMP) [SSMP85] is a protocol for efficient, half duplex communication across a data network between an application program on a host computer and a remote character mode terminal. The system configuration is shown below.

![Schematic view of the SSMP protocol over a PSN.](image)

The terminal must have a processor with sufficient capacity to support the protocol in a position logically close to the terminal in the network structure. Such a processor could be a micro computer or a terminal concentrator. The objective of SSMP is to minimise the data transmitted across the network between the host and the terminal, improve the responsiveness or quality of service to users, and to minimise the amount of processing done by the host computer by reducing the number of host "wakes". Host wakes are directly related to the number of packets sent to the host, the host wakes every time data from the terminal arrives in its input buffer.

The program running on this processor is called the "terminal emulator" or the "terminal manager". This is an "intelligent front-end processor" to the host computer which performs simple editing functions, cursor movements, interpretation of user keystrokes, and a number of other functions locally to the terminal as specified by a host application. Functions that cannot be processed locally at the terminal manager are passed back to the host application program for processing in a packet of encoded data preceded by the previously batched data from, locally processed commands. This causes the host to wake up, process the backlog of commands and send packets of commands back to the terminal. The commands and data in these packets are specially encoded to minimise the data sent across the network.

The terminal manager processes user keystrokes by examining these and translating them into "logical keystrokes". This process maps some special function keys into a standard set of logical
3.2 SSMP Description

keystrokes. This partially isolates the host application program from the physical characteristics of the terminals with which it communicates, and produces more portable applications programs.

The protocol has two modes of operation:

- Line mode
- Screen management mode.

*Line mode* is "normal" terminal operation. Lines of characters are entered by the user to form commands and sent directly to the host when the carriage return is entered. The terminal manager is dormant during line mode, it only scans the data for a special initialisation message. A simple example of command mode is a line-mode editor.

*Screen management mode* is a fully interactive mode where individual user keystrokes have significance on their own. This is the mode of a full screen editor. The time while the terminal manager is in screen management mode is called a *screen management session*.

A screen management session is initiated by the host application sending a special initialisation message to the terminal. This is understood by the terminal manager which then expects the host application to bind a number of functions to user keystrokes. This involves some negotiation until a suitable, mutually satisfactory binding is found. All of the functions and other parameters set in the terminal manager are chosen by the host application. During a screen management session, both the terminal manager and the host application maintain a "shared data structure" (SDS). This describes the state of the session and of the current screen image. The shared data structure contains the following items:

- access token
- stored screen image
- cursor position
- mode parameters
- update fields
- tabs, etc...

The *access token* is used to enforce the half duplex operation of the protocol by allowing only the current holder of the token to update the shared data structure. This access token can be visualised as a physical key that is passed back and forth between the host application and the terminal manager. The terminal manager will only accept user keystrokes when it has the token, otherwise it will queue them until the token returns from the host. When the host has the access token, it sends commands to the terminal manager which queues user keystrokes and processes these commands to update the shared data structure.

The stored *screen image* is a matrix of characters that describes the data currently displayed on the screen. This allows the terminal manager to access the data that the physical terminal is displaying.

The *cursor position* description defines the current cursor position on the screen. This is a
simple x-y coordinate of (row, column). The rows and columns are numbered from zero in the top
left corner.

Mode parameters are used to describe the state of a number of session parameters. These are
simple values that describe the state of features such as the keystrokes bound to terminal manager
functions, etc.

The update fields define regions of the screen outside which no alteration to the screen image
can take place. Thus all editing is restricted to within these rectangular limits. These are defined
with a 4-tuple describing the top, bottom, left, and right of a "box". These can also be thought of
as fixed size, non-overlapping windows on the screen.

SSMP is labelled "simple" because of a number of simplifications made in its design. The
protocol provides support for:

• One virtual screen per physical terminal
• Standard ASCII terminals
• Rectangular bounds on user cursor movements
4. IMPLEMENTATION.

4.1 Implementation Decisions.

This chapter describes some aspects of the experimental implementation of SSMP built for this evaluation of the protocol. The host program that was used to test SSMP was the Chef text editor. This was chosen because of its availability and simplicity which allowed for easy modification. The "screen mode" of Chef, a full screen editing facility, was well suited to SSMP because of its simplicity and the way its functions operated. Thus, Chef made a very good tool for testing the protocol.

A number of decisions had to be made to simplify and clarify the implementation. The Fawn book [SSMP85] describes, in great detail, the algorithms to be used, and other functional capabilities of the protocol, but it leaves some of the technical aspects of implementation to local interpretation. This section describes the most important simplifications and design decisions made for this implementation. A listing of program listing of the experimental model may be found in appendix J. The features considered were as follows:

- Compatibility with only one terminal
- Perfect communication channel
- Reduced shared data structure

The terminal manager has been designed to communicate with only one type of terminal. Because of the variation in physical terminal characteristics and the code translation required to produce logical keystrokes, it was a huge simplification to restrict the design to the "Televideo model 925". This terminal was chosen because of its availability, not for any reasons of compatibility with SSMP.

A perfect communication channel between the host computer and the terminal manager was assumed. This simplified the initial design, since all packets sent down the channel at one end could be guaranteed to arrive intact at the other, so no error checking was required. Later the communications channel was modified so that errors could be introduced in a controlled manner. This allowed testing of the behaviour of SSMP under various situations.

The screen image was not included in the host's copy of the shared data structure to simplify the interface and increase the efficiency of the host interface. If the host requires a stored image of the terminal screen, it should manage its own data structure. SSMP recommends that the host program maintain a stored copy of the screen image as well as the terminal manager, but it does not make this feature compulsory. Because of the processing required by the host computer to maintain this data structure, it was decided to completely omit it from the shared data structure.
4.2 The Simulated Data Channel.

Why Simulate?
A simulated data channel was used to transport data between the host computer and the terminal. The reasons for not using a real data network are as follows:

- A real data network was not available
- Connection to a real network can be fraught with complexities
- Channel parameters can be changed easily with a simulation
- Line mode communication could be ignored by the terminal manager
- Simulation provides a much more reliable and easily managed system

The channel parameters that can be altered for the simulated model are the propagation delay, the error rate, and channel capacity, this gives a very high degree of flexibility in an evaluation.

Communicating through pipes.
The host application and terminal manager were built as independent, asynchronous processes running within a single computer. The channel was simulated using "pipes" in the UNIX™ environment. One pipe was constructed to carry data in each direction between the two components of the system. Because of the complete independence of the two processes, they are often referred to in the remainder of this report as if they were actually physically separated.

The independence associated with the pipes ensures that the "rules" of the protocol are followed. It would have been very easy to implement the simulation with a single program and to communicate through an internal buffer system. This would have allowed the possibility of subtle errors in the model. For example, communication between the parts of the program via global variables.

When communicating with the terminal in screen management mode, the host process communicates all data and control information with the terminal through the pipes. During line mode operation, the host application sends data directly to the physical terminal, by-passing the terminal manager process. This simplifies the terminal manager a great deal as discussed in the following section.
4.3 The Terminal Manager.

The main algorithm.

This section discusses the general structure of the terminal manager program built for this project. The terminal manager has been designed to be as modular and portable as possible within a strictly layered structure to allow for easy modification. The major functions performed by the terminal manager are:

- Keystroke interpretation
- Cursor movement
- Screen image manipulation
- Communication with the host

The main program of the terminal manager, after constructing the appropriate "pipe" connections, controls the basic operation of the protocol. A highly simplified pseudo-code description of this algorithm is given below. The section between the lines shows the input scanning algorithm for a real network implementation. Using the simulated network, this section was not required since all line mode communications were sent to the physical terminal directly from the host, by-passing the pipe connections and the terminal manager.

```
  terminal_main :
  begin
    forever do begin
      { ---------------------------------------------
        while (access_token_at_linemode) do begin
          get_input_message;
          if (startup_message_found) initialise_session;
          else display_message;
          send_any_user_keystrokes_to_host;
        end;
        -------------------------------
        while (access_token_at_host) do begin
          if (input_buffer_empty) then receive_packet;
          get_token_from_buffer;
          process_token;
        end;
        while (access_token_at_terminal) do begin
          get_logical_keystroke;
      }
    }
  end;
```
4.4 Implementation

```
interpret_keystroke;
if (output_buffer_full) then send_packet;
end;
end; {forever}
end;
```

It is convenient to view the access token as being in one of three states:

- at the host - the manager processes host tokens, user keystrokes buffered
- at the terminal - the manager is processing user commands
- in line mode. - the terminal scans the input for an initialisation message

The token is used to indicate the current state of the manager. Initially the access token is "at the host". This allows the host process to initialise the terminal manager.

The routine "receive_packet" gets the next packet from the input pipe, if there is no packet waiting to be read in the input pipe, the terminal manager will sleep while it waits for a packet to arrive. Thus pipes are read and written in blocks that form packets during the transmission.

To end a screen management session, the host issues a command token to cause the manager to release the terminal from screen management mode, and to set the token to "line-mode". For this implementation, this involves returning the token to "at host" and sleeping while waiting for the next session initialisation packet to arrive in the input pipe.
4.4 The Host Application Interface.

The communication toolbox.

The major task of the host application interface is to communicate with the terminal manager. This requires the correct encoding of data as a mistake may cause severe errors at the terminal. It was decided to provide a toolbox of routines (not specified by the protocol) to perform the complex tasks of reading and writing the specially coded data to the data channel. This relieves a lot of the burden on the applications programmer and can allow local variations in interpretation of the standard while still allowing an easily programmed interface. The toolbox output functions provide all the functions specified in [SSMP 85] for the host application to signal the terminal manager.

```plaintext
toolbox_output_function :
  begin
  update_data_structure;
  write_tokens_to_output_buffer;
  if (output_buffer_full) then send_packet;
  end;
```

A call to the terminal input function "tp_get_screen" will cause the program to sleep until a packet arrives at the input pipe. This is then automatically copied into the input buffer and the first command token returned. The function "tp_get_screen" acts as an intelligent, selective decoding and input routine. The basic algorithm for the main input routine is as follows:

```plaintext
tp_get_screen:
  begin
  repeat
    if (input_buffer_empty) then receive_packet;
    get_token_from_buffer;
  until (token_is_valid);
  end;
```

Host Application Program Modification.

The Chef text editor is a well structured modular program with one module (file) in which all the "screen mode" terminal interfacing is done. Extensive modifications to this module of the editor had to be made before Chef could communicate with the terminal manager using the host toolbox. Some additions for translation between the Chef and SSMP data formats have been added, but most of the alterations have been deletions of functions now done by SSMP.

SSMP could not perform some of the functions required by the screen mode of Chef which
included *macros* and *automatic word wrapping*. It is not possible for SSMP to support these functions at all in its present form, which is a severe restriction. This problem and others will be discussed in the following sections.

Unfortunately, Chef is so well suited to the way the protocol works, that it does not test the protocol's ability to adapt to different situations. However even from this simple implementation, several problems with SSMP, especially its inflexibility, were demonstrated. The Chef editor with SSMP operates as follows:

*Fig. 4.6: Schematic view of Chef using the SSMP protocol.*
5. EVALUATION.

5.1 The Users' view.

This section discusses the advantages and disadvantages of SSMP from the point of view of a
user of an application. The service to users is a very important aspect of the protocol, for if SSMP
cannot maintain favour with its users, its use will decline. The protocol in its standard form has
only one advantage

- Response time for many user commands is reduced.

There are, however, many more disadvantages, as follows

- Large variations in response times confuse users
- Application functions often require modification to use SSMP
- Users require knowledge of the terminal manager

Fortunately for SSMP, the one advantage it offers over a directly connected terminal at least
balances in importance with the disadvantages. Of all the factors listed above, the response time is
of most concern to users. Thus by decreasing the average response time, most users will be
pleased. The disadvantages discussed below are not substantial problems, and generally do not
have a very large effect upon the users view of the protocol.

Variations in responses.

User commands that are serviced in the local terminal manager will usually have a much
smaller response delay than those processed at the host site. If the local implementation of the
terminal manager is made on a micro computer that is dedicated to the user, then locally serviced
functions will be extremely fast. If the terminal manager is implemented within a local terminal
concentrator, then the service to the user will depend on the number of active terminals connected to
the concentrator and the processing capabilities of the concentrator. If a poorly functioning
machine is used to host the terminal manager, it is possible that there will be a decrease in the
performance of the remote applications run by the users.

Functions that require the host program to perform some processing will take longer to respond
than they would for a directly connected terminal. This is because a packet containing all of the
backlogged data from locally processed commands must be processed before the host request is
serviced. Additional delays are contributed by the encoding and packet assembly done by the
protocol, and if the network is congested or the host machine has a high workload, then the delay
before a response packet returns from the host will be very large.

The most common problem arising from having a large variation in response times for a group
of commands is amplified by the impatience of most users. If, for example, a command takes
between one and six seconds to respond, then after a user enters the command and has waited for two seconds, he may begin to believe that he has not entered the command correctly since he has not witnessed a response. He may then proceed to enter the command again. Thus too many commands have been entered, and it may not be until much later that this error is detected. This problem is made even worse by partially processing some commands at the terminal manager to update the screen image say, and then signalling the host to update the appropriate data structures. In this case, if a host processed command is entered a number of times in succession (intentionally), the effect of the first of these will be seen very quickly due to the partial local processing, but the remaining commands will be preceded by a lengthy delay. This again may cause the user to enter more commands than required, since, due to the delay, he may believe that he has entered a number of the commands incorrectly. This partial processing is demonstrated in [Macl82].

Thus from a user's point of view, the first objective for designing an SSMP interface for an application is to design commands to have the most consistent response times possible. It is often very difficult for commands that are processed at the remote host to respond with consistent delays because of network delays and host computer workloads that may vary a great deal throughout a day. Such aspects as this are beyond the control of both the user and the designer.

SSMP allows a command preprocessing scheme to be used, but this is not enforced. The protocol leaves the decisions described above to the host application program. Thus the variation in response times is generally determined by the way the host application uses SSMP and not upon SSMP itself.

**Modifications to applications.**

Some application's terminal interfaces may have to be modified to accommodate SSMP. This may restrict the commands available for some applications, or make some user commands different from those of directly connected terminals. The effect of these alterations may decrease the popularity of some applications software within the user community. It is very important for the protocol to operate transparently so that the user does not see any differences in his applications.

SSMP requires more flexibility in this region to allow applications to specify the actions of the terminal manager to a much finer degree. At present an application must conform to the rigid structure enforced by the protocol.

**Knowledge of the terminal manager.**

A further problem that a user of SSMP may encounter arises when an error occurs. This may be an unexpected message arriving at the terminal (a very common occurrence with system messages, etc.) or, much more rarely, a hardware or software fault. Such an error will cause the protocol (screen management session) to revert to user control. Whenever the user takes control of the protocol, two options are available. The session can either be suspended (for later restart) or interrupted. Most commonly the session will be interrupted and the user will begin again. Thus
the user must be aware of the presence of the terminal manager and have the knowledge to cope with any exceptional conditions.

It is thought that it is impossible, to provide any protocol of this type that behaves well in an error situation. Hiding the operation of a communications system when an error occurs is a problem with many distributed computing systems that has no simple solution. The simplest and most user friendly way to overcome this problem is to provide a single "panic button" that escapes from any undesirable situations that may arise and leaves the user in a safe, familiar position. From here the user may initiate any recovery or tidying operations required. The exact effect of the panic button at any time should be defined by the host program, but a general default action should also be available.

SSMP handles manual session control in a satisfactory way, but again the method used lacks flexibility. The operation for "interrupting" the host is very similar to the "panic button" described above, but the action of this function can not be defined or controlled by the host application.
5.2 The Applications Programmers' view.

This section discusses the disadvantages of SSMP as seen by an applications programmer. The only advantage of the protocol, as seen by a programmer, is:

- Partial isolation from the physical characteristics of different terminals.

A major problem with the protocol is that it does not provide a simple, easily programmed method for interfacing with the terminal. Thus the programmer is faced with a great deal of work to implement an application using SSMP. The major disadvantages are:

- The complex communication scheme used
- There are many possible outcomes for a series of function requests
- Software may be designed to "fit" the protocol.

Isolation from terminal characteristics.

SSMP provides some standardisation of physical terminal characteristics. This is done by mapping specific terminal interactions into "logical keystrokes" at the terminal manager. This allows the programmer to ignore the specific characteristics of the physical terminals. The standardisation of keystrokes is a very important feature of the protocol because of the associated increase in the portability of applications programs.

The mapping scheme used by SSMP has some faults as discussed later in this report, but in general, the protocol provides a satisfactory level of isolation in theory. In practice, a very vague description of the mapping to be performed is made ([SSMP85]) which may lead to variations between the schemes used at different installations. This removes most of the advantages of having a standardisation scheme, because applications programs will have poor portability.

Complex communication scheme.

The applications programmer is faced with a very complex task to design and build a front end for a program that conforms exactly to the SSMP standard. The standard relies very heavily on the host application program "knowing what to do" in any situation. For example when a binding of a function to a keystroke occurs. For this reason the terminal interface of any applications program that wishes to use SSMP must be very large and complex. This places a great deal of responsibility on the applications programmer to predict all of the possible situations that may arise, and to cope with these correctly.

The complexity of the task of building an application program interface using SSMP may result in the protocol falling into disuse because of the work required by the programmer to make the system operational. Many programmers will find it much easier to address terminals directly than with SSMP because of the reduction in the complexity of program design. SSMP specifies no standardised host application interface or toolbox to assist the programmer in any way.

For any protocol of this nature to gain widespread use, it must be designed to allow very
portable implementations of applications (very strictly standardized) and it must be able to be implemented quickly and easily by a large population of programmers.

**Function binding alternatives.**

SSMP allows some terminal manager functions to be optional, to take full advantage of the built-in features of a modern terminal and also be able to communicate with very "dumb" terminals (those with extremely limited capabilities). This idea leads to a number of complications for the programmer, especially with portability. When a screen management session is initialised, the host application tells the terminal manager which functions the manager is to perform ("set-mode" commands [SSMP85]). The terminal manager then has to reply to these requests with either a positive acknowledgment, in which case the host assumes the manager is capable of performing the function, or a rejection of the request. Upon receipt of the rejection, the host application program has to change its strategy and process the rejected function itself.

Thus for many of the functions the host application wishes the terminal manager to perform, the host must have multiple alternative plans of action depending on the reply to the function availability request. This is an undesirable situation since, if a refusal is not handled by the programmer, the program may not operate at all if a less capable terminal is connected to the system. Thus for n functions to be handled by the terminal, the programmer must have designed an action for the $2^n$ possible combinations of outcomes which would be a large and time-consuming task. A far better scheme from the programmer's point of view is to have a reduced set of terminal functions and a guarantee that all terminal managers that obey the protocol will provide all of these functions by whatever means possible.

The programmer should be presented with a clear, simple, well documented method to communicate with a terminal without having to cope with a large number of subtle details and possible alternatives. From the programmer's point of view, SSMP is not likely to be used by choice because of the complexity of the task of handling the possible outcomes. If the programmer is not happy with a protocol, a new one is designed and the old is quickly forgotten !.

**Designing Software to fit SSMP.**

A further trap that SSMP may draw a programmer into, is that of designing new software to "fit" the protocol. This may assist users since many applications will behave similarly, but the functionality of the new applications may be limited within the restrictions of the protocol. This problem is typified in [HH82] where an editor is designed to optimise the performance of a screen management protocol.

All screen management protocols will probably have this effect to some degree. To minimise the problem, SSMP should supply a more generalised set of functions that allow applications more freedom, while still making good use of the protocol.
5.3 Keystroke Interpretation and Binding.

This section discusses the methods used by SSMP to represent and manipulate user keystrokes and to bind these to terminal manager functions. SSMP has a very restrictive scheme for classifying user keystrokes. The user may enter either:

- Data (printable ASCII characters and escaped commands)
- Commands (control characters, escape sequences, and special keys)

The inflexibility of this scheme leads to several problems. The binding of user keystrokes (commands) to terminal manager functions is also very restrictive which also produces difficulties. These issues are discussed below.

Escape key restrictions.

The method used by SSMP to implement "escape" sequences of keystrokes is very restrictive. Escaped printable characters are always interpreted as "commands" by the terminal manager, while escaped control characters are interpreted as data, and displayed as the built in "replacement character" ('~'). This control is built into the protocol and cannot be changed. The rigidity of this scheme may be incompatible with some applications and will not allow some applications to use the escape key as a function key by itself. This is unsatisfactory for the user especially, who may become confused with a reassignment of function keys. SSMP should allow the escape key to be used for any function as defined by the host application. This could be done by reassigning the function of the escape key to another key, so that "escape" can be interpreted as a single keystroke.

Keystroke representation.

The representation of user keystrokes is one of the most important features of a screen management protocol. This involves translating the physical terminal keystroke codes into a standardised representation or logical keystroke so that the host program can operate completely independently of all of the physical terminals with which it may communicate. SSMP performs a limited amount of this standardisation, the major faults with this are:

- No strict specification of the mapping of special function keys.
- The following keystrokes are indistinguishable (esc/a, esc/A)
- The logical keystrokes overlay the set of possible keystrokes. This means that two keys may be represented by the same logical keystroke code, and therefore the host application cannot distinguish between these. For example, if the host wishes to use both of the commands "ctrl/K" and "cursor up", this is not possible. This is because SSMP maps the cursor key to the control sequence "ctrl/K", so the host cannot distinguish between these two events. This limitation is quite unacceptable when there is a large number of special function keys.
- Cannot determine the name of a logical keystroke for displaying in menus and other user
5.7 Evaluation.

information devices. This is a severe problem because logical keystrokes can be bound to functions by the host, but there is no way for the host to inform the user of what the physical keystrokes that invoke the bound functions are. If the logical keystroke names are displayed on a menu, the user will require a keyboard overlay to determine which physical function keys produce the logical keystrokes displayed.

**Binding keystrokes to functions.**

The method used by SSMP for binding user keystrokes to terminal manager functions is very rigid and restrictive. The binding used is a "one to one" relationship between one of the terminal manager functions provided and the keystroke that invokes it. This restricts the host application to use either a terminal manager function in exactly the way it has been implemented by SSMP, or not to use it at all. To activate a function, a logical keystroke code is directly associated with the function. This method has many restrictions that limits the number of applications that can use the terminal manager as follows:

- The built in functions of the terminal manager often differ slightly from the functions required by an application, rendering them unusable. This reduces the efficiency of an implementation because the host has to process many more functions than it should have to.
- Combinations of functions cannot be bound to a keystroke to produce "custom built" functions for an application. For example, the "cursor up" and "cursor right" could be combined to produce an function that moves the cursor diagonally. This has the same effect as the previous point.
- A keystroke will, in general, have the same meaning wherever it is used during a session, unless its binding is specifically changed. This is not efficient when an application has several "modes" that each have a different command set.
- Keystrokes that are bound to functions may only be "commands", as determined by SSMP, not by the host. Often applications have a command mode where data characters can be used as commands. This mode of operation is not supported by SSMP.
- Functions can only be performed by explicit entry of a function-bound keystroke. Implicitly invoked functions such as automatic "wrapping", where the function is invoked by exceeding a given column, cannot be performed under this scheme.

To summarise these criticisms of SSMP, the method used to bind user interactions with terminal manager functions is too inflexible. A sufficient degree of specification to satisfy the needs of many applications cannot be made with SSMP. This method of keystroke interpretation and binding is insufficient and is the largest fault with the protocol.
5.4 Terminal Manager Functions.

The terminal manager provides a number of standard built in functions that can be invoked by the user if the host application program binds a keystroke to them. It is difficult to predict what functions will be useful to most applications and for this reason it is very tempting to add a very large number of functions to the manager. Since the manager is designed to operate on a small local processor, it is desirable to keep it as small as possible. However as many functions as possible should be provided that perform operations that result in large savings in the amount of data traffic and host processing required, such as cursor movement functions. A list of the functions provided by the SSMP terminal manager is given in appendix B.

It is generally undesirable for the terminal manager to provide "high level" functions or functions that perform complex tasks. This is because every application has a different method for achieving a similar result. Thus, if a small series of high level functions is implemented, it is unlikely that their actions will be exactly those required by many applications, thus the function will seldomly be used. The terminal manager features provided by SSMP are discussed below. These are divided into five categories as follows:

- session control functions
- cursor movement functions
- in-line image editing functions
- inter-line image editing functions
- mode parameters

Session control functions.

These are special functions to allow the user to manually control a screen management session. They are particularly useful if a transmission error or a hardware or software failure occurs. The protocol provides functions to suspend and restart sessions as well as to interrupt (cancel) a session. This scheme is satisfactory, but difficult for the host program to control. The host binds keystrokes to these functions, but it cannot dictate the exact outcome of these functions because it is not notified when they occur. Minor control over the actions of the interrupt control could be possible if the host program "trapped" the interrupt signal sent from a host interrupt command. This is very unportable and may often be very difficult to achieve (not recommended). The suspend and restart functions may be useful in some circumstances, but it is thought that these will be mostly ignored because of the knowledge required by the users to understand how to use these.

SSMP has provided a satisfactory set of session control functions, but these could be improved by allowing host applications to define the actions of the session controls more precisely if they wish.
5.9 Evaluation.

**Cursor movement functions.**

These SSMP functions simply reposition the cursor at a new coordinate on the screen. They pass nothing back to the host program until an image editing function is performed. At this point the new cursor position is sent to the host immediately before the image editing information. If the cursor has not moved since the last function, the cursor position is not sent since it is already known in the shared data structure. This scheme is very efficient in terms of both data sent and processing required. Since these functions are very simple and produce such a great saving when processed by the terminal manager (no traffic to the host at all) then it is very advantageous to provide as many of these functions as practically possible. The SSMP terminal manager has a rich set of such functions but more could be provided to improve its flexibility.

Overall, the SSMP cursor movement functions operate very well. They are very efficient in terms of the processing required by the host and the data transmitted across the network. As discussed in section 5.3, a method for combining these functions to produce complex customised operation would improve the functionality of protocol a great deal.

**In-line image editing functions.**

These are functions that alter the data in the screen image within one line. Included in this group of functions are "insert-space", "delete-character", "erase-line", etc.. These are simple functions that do not usually require the host program to be notified. It is advantageous to provide terminal manager processing of these functions because they occur very frequently and have quite minor effects on the screen image so the host usually does not require notification.

The most common problem with the SSMP in-line functions, that often makes them incompatible with host applications, is the actions they perform at the edges of the update fields. An example of this is the delete character function when invoked on a line of data that is wider than the current update field. When a character on the current line is deleted, the characters on the line shift to the left by one character position moving a blank to the far right position. The host application may wish to display the character shifted in at the right side of the field for lines wider than the current update field, but it cannot detect this requirement unless it is notified for all the "delete-line" functions performed. In this case the host could process the entire function. Similar problems arise with the "erase-line" and other commands. Thus the way in which SSMP handles in-line functions is unsatisfactory in some instances such as with Chef.

The in-line functions of the terminal manager lack flexibility in the same way as the cursor movement functions do. A more general approach is required to meet the needs of a wider range of applications. This could involve the addition to SSMP of a method for signalling the application program in certain circumstances, such as described above.

**Inter-line image editing functions.**

The SSMP image editing functions are "line oriented". Operations on lines (inter-line functions) move the lines of the screen image relative to each other. This often requires the host
application to be notified to update the screen image of the terminal manager.

It is difficult for a text editor such as Chef to allow functions such as "delete-line" to be handled by the terminal manager. If this was handled by the manager, Chef would not be signalled to display any new lines at the bottom of the screen after a delete-line function. Because in this case the inter-line functions has to be processed by the host application anyway, any use of the inter-line functions in the terminal manager was unnecessary.

It is assumed that SSMP has included the inter-line functions at the terminal manager "just in case" an application wishes to use them. These features encourage application implementors to use the functions to perform all the local processing required for a command, while also signalling the host program to alter its data structure. This method of sharing the processing for an operation between both processors is not at all beneficial. The effect of this partial local processing is that a screen image update begins immediately but takes a long time to be completed due to the host processing. This causes problems for the user as discussed in section 5.1.

**Terminal manager parameters.**

The SSMP terminal manager also provides a collection of "mode" parameters that may be set by the host application or by the user (under control of the host application). These parameters describe the state of a number of features of the terminal manager. The parameters are listed in appendix B.

It is unlikely that some of these parameters ("ilinerow", "cursor", etc.[SSMP85]) would be used by many applications. However the presence of these mode parameters has no effect on the efficiency of the protocol. There are, however, many functions that SSMP cannot perform at present that could be added to the protocol easily with these parameters, such as a mode where all keystrokes are signalled to the host. The protocol would be improved by adding more of these parameters to the shared data structure to increase the number of applications suited to the protocol. However, the standard set of parameters provided by SSMP does perform many of the functions that are required.
5.5 Data Link Management.

Packets in the channel.

Controlling the communications channel was very simple using the "pipes", allowing the half duplex operation and other features of the protocol to work very well. The maximum packet size used for the communication was altered to see what effect this had. The maximum packet size recommended by [SSMP85] was 60 characters, however, this was found to be very short when the data channel was examined using Chef, so the packet size was experimentally expanded to 256 characters. Appendix E shows the results of the some measurements on the data with the new packet size. These results show that on average the packets are quite small, but at session initialisation, some very large packets are send, constrained only by the maximum packet size. Some examples of the packets sent are given in appendix F.

This indicates that a more efficient protocol will result from allowing a larger packet size. SSMP gives no reason for choosing a packet size of 60 characters, so it seems that the larger size should be considered. The standard packet size must be adhered to, however, if the applications that use SSMP are to remain portable.

Data encoding efficiency.

The data encoding algorithm used by SSMP is reasonably efficient in terms of the data sent, but encoding and decoding the data at either end of the communication channel requires a non trivial amount of processing to implement the SSMP encoding algorithms. An important feature of screen management protocols is the reduction of processing to be performed by the host computer, so the data encoding and decoding algorithms should be as simple and effective as possible.

The encoding scheme specified by SSMP requires many printable, non alphanumeric characters to be encoded as two character sequences. This scheme results in a greater volume of data being sent across the network than necessary. The justification given for this is that some printable characters may have a special meaning to some networks. This is a very cautious attitude, and is thought to be quite unnecessary in almost all modern data networks.

Some command tokens have repeat counters attached to them. This is an integer that indicates the number of successive times a command was invoked at the terminal. It is often uncommon for a command to be entered a number of times in succession, thus the repeat counter serves little useful purpose (always "1") but results in a larger volume of data transmitted, and additional processing to maintain the scheme. SSMP is insistant that this scheme reduces the data sent across the network. For Chef and similar editors where a function is not frequently entered many times in succession (except possibly for the two "insert-char.", and "delete_char."), the averaged effect produces more data sent than necessary.
Testing under different conditions.

The simulated data channel was used to vary the transmission delay of the data channel so that the effect of the locally processed functions could be seen. This was useful to gauge user reactions to the variations in response delays. Transmission errors were also simulated to see how SSMP handles these. The method for detection and recovery from errors used by SSMP is very simple, but it works well, unless an error occurs within a packet. The protocol has no way to handle this situation at all. It assumes that the contents of a packet will not be distorted, but that miscellaneous transmission may arrive between packets. In general this assumption holds true because of the buffering of the packets and the error free data channel (error handling performed by lower layer protocols eg. X.25).

SSMP passed all of these tests adequately, the delay test showed how the host application has complete control over the timing of most commands, and clearly demonstrated the problems with partial processing at the terminal manager (command begin very quickly, but take a long time to complete). The error detection is handled by SSMP as well as it can be.

It was also possible to vary the buffering capacity of the channel, but this was not done. It was clear that the protocol had no way to restrict the number of packets that may have to be buffered at either end of the channel. It seems that to guarantee a successful session, the protocol requires some form of flow control. This could be done with a complex algorithm with data in each packet to inform the other station of the state of the local buffer, or by guaranteeing a given buffer size that may not be exceeded. The second method is preferable for reasons of efficiency and simplicity.
5.6 Performance with Chef.

The protocol performed most of the functions required by Chef at a similar speed to the original directly connected version. A speed increase in the terminal interaction was not expected since the implementation was done within a single machine as described in section 4. Formal speed comparisons could not be made since it could not be determined how much of the total transmission delay was spent transporting data through the pipes. It was also unfair to compare the speed of the implementation with the original version of Chef since no parallel processing of the terminal manager with Chef could be performed as in a real distributed SSMP implementation, i.e., the two programs were competing against each other for resources of a single computer. The following two of the functions of Chef could not be implemented at all using SSMP.

- automatic line wrapping.
- macro definitions.

These functions are not difficult to implement and commonly occur in text editors and other applications. The unique problem with these functions is that they cannot be performed by the host program if the manager does not support them. Because they are not activated by specific user entered keystrokes, the SSMP terminal manager can never detect them. The automatic wrapping function is triggered by the user entering a data character (other than the blank character) to the right of a specified column, while macro definitions require every keystroke entered to be acted upon and then stored in the current macro definition buffer. Macros are difficult to implement in a standardised way in the terminal manager because most applications have their own unique methods for controlling these.

The most noticeable factor when using the SSMP version of Chef was the delay to start up "screen mode". The time taken for Chef to initialise the terminal manager was quite substantial, but this is a property of all screen management protocols. It would be beneficial to reduce this startup delay to increase the productivity of impatient users. However, this time delay is so consistent, it causes no problem to the user who adapts to expect it.

The functions "split-line", "insert-line", and "delete-line" were implemented in two ways to gauge the value of their incorporation into the terminal manager's set of built in functions.

- To take full advantage of the terminal manager functions
- To ignore the manager's functions and do these from Chef.

When the manager's functions were used, a problem became apparent. Whenever Chef inserts or deletes a line from the screen image, it updates an index at the top of the screen to indicate which lines are displayed. By processing the inter-line functions locally to the terminal, Chef could not update this counter at the correct time. When a packet was eventually sent to Chef to inform it of the alterations, the counter on the screen was rapidly updated to the latest situation again, confusing the user. This situation was remedied by using an SSMP parameter that forces certain
5.14 Evaluation.

functions to notify the host immediately. This worked well, but it was obvious that since Chef was waking up, inserting a line in its data structure, and updating the screen anyway, that it would be best if it performed the entire function. Thus the second implementation was preferred, since it performed as well as the first, and was much simpler.

The performance of SSMP was measured with Chef. The data channel simulation allowed the data transmissions to and from the host to be measured easily and accurately. The results of a number of experiments is given in appendix E. To summarise the major results:

- Average host wakeup improvement = 25
- Average data transmission improvement = 7

The average host wakeup improvement is the factor by which the number of host wakeups has been reduced by using SSMP. This is averaged over a series of experiments.

The average data transmission improvement is the factor by which the volume of data sent across the network is reduced for data entry experiments. This factor assumes that an X.25 packet switching network is used.

Thus SSMP has improved the terminal interaction efficiency of Chef substantially. However, the functionality of Chef has suffered to achieve this improvement.
5.7 Summarising SSMP.

This chapter has highlighted the major aspects of SSMP. From this critical evaluation, the following advantages and disadvantages of the protocol have arisen.

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Faster user responses</td>
<td>Large variation in response times</td>
</tr>
<tr>
<td>Less data sent through the data channel</td>
<td>Terminal manager not always transparent</td>
</tr>
<tr>
<td>Less host processing required</td>
<td>Difficult to built application interfaces</td>
</tr>
<tr>
<td>Partial isolation of terminal characteristics</td>
<td>Very inflexible</td>
</tr>
<tr>
<td></td>
<td>Data encoding not very efficient</td>
</tr>
<tr>
<td></td>
<td>Lack of absolute standardisation</td>
</tr>
</tbody>
</table>

Of these, all of the advantages are common to most screen management protocols, but only some of the disadvantages are. The major disadvantage, inflexibility, is unique to SSMP (this is only of concern to general purpose protocols). The inflexibility of SSMP are in the regions of:

- Keystroke representation
- Binding functions to keystrokes
- Terminal parameters (modes)

This restricts the applications that use SSMP to either conform to its conventions, or not to use it at all. Many functions required by applications either cannot be implemented in their original forms, or they cannot be implemented at all. Many of the basic concepts behind the protocol are good, but the details of the functions specified by the protocol are often implemented in inefficient ways.

SSMP is also disappointing for its lack of features for interfacing with the latest technology equipment. The protocol lacks any method for addressing a mouse or other pointing device, and provides no facilities for handling "soft buttons", and many word processing functions such as automatic line wrapping. This further restricts the range of applications for which SSMP is suitable.

The situation to which SSMP is best suited, is to implement a modified version of it as a customised protocol for a specific task. This would produce a very unportable, but highly efficient protocol suited only to a limited number of applications. As a general purpose protocol, SSMP fails to provide a sufficiently high level of generality to satisfy many applications. It should be fairly easy to design a more general protocol to "fix" many of the problems found with SSMP. The following sections look at how this might be done.
6. DESIGNING A BETTER PROTOCOL.

6.1 What is required?

From the evaluation of SSMP it should be clear that this protocol is not adequate for a large number of applications that may wish to use it. The major fault that has been found with SSMP is the inflexibility of the protocol. It is clear that a protocol of this type should provide a much more flexible or generalised method for the application program to bind the manager functions to user interactions. There are other areas also where SSMP can be improved. This section attempts to define exactly how a screen management protocol should behave, and what facilities are required to allow a host program to strictly control the manager so as to provide an efficient, transparent service to the user.

The ideal features of a screen management protocol are as follows:

- A large set of "primitive" functions at the terminal manager
- All terminal manager functions compulsorily implemented (by whatever means)
- The ability to bind one keystroke to a sequence of functions
- The ability to bind one function to a number of keystrokes
- The ability to bind functions to events other than user keystrokes
- A macro facility
- A method for tracking a mouse or other pointing device
- Overlapping windows
- Optional variable size characters, and abstract graphic symbols.
- A general method for manipulating white space (blanks, tabs, microgaps, etc.)
- The ability to use the escape key as a function by itself
- A mode where all keystrokes signal the host (command mode)
- Simple, transparent session control
- Minimal data transmitted between the terminal and the host
- Minimal processing required to encode and decode data
- Highly packetised data transmission
- Maximal pre-processing at the terminal manager
- Easy for programmers to implement applications using it.
- Independent of specific terminal characteristics
- Multiple virtual screens
- Communicating with multiple hosts simultaneously (multi-tasking)
- Compatibility with Postscript and other modern methods of data representation
Although the efficiency of a general purpose screen management protocol is important, it is less important than its flexibility. The amount of data transmitted between host computers and remote terminals is often of little concern to the user and the network since most modern data networks have very high capacities and charge low rates for the data transmitted (often the annual connection fee is greater than the data volume fee). Thus the primary requirement for a screen management protocol is to deliver the best possible (fast and transparent) service to the users of the system.

A further requirement that can reduce costs and again increase the speed of service to the general user community is to minimise the workload on the host computer. Also by reducing the number of interruptions to the host computer by batching input data from the terminal in packets, the host will be allowed to perform more useful work by not wasting as much time swapping processes to service terminal requests.

It is impossible for any protocol to support all applications, but most simple text editors and other very commonly used, highly interactive applications should be able to employ the protocol. The functions provided by the protocol, therefore, should allow the most common features of these applications to be implemented as efficiently as possible without changing the way that they are used. Thus the terminal manager needs to support a facility for binding functions to user interactions in a very flexible way so that applications can customise functions for their needs.

**Alterations Required by SSMP.**

Many of the features listed above cannot be added to SSMP without major structural changes to the protocol. The major problem when implementing the Chef text editor with SSMP was that there were no manager facilities for performing some of the editing functions required. The most obvious and simplest solution to this problem is to add some additional functions to the manager. All of the other alterations required by SSMP to allow it to perform as described in the previous section would involve basic structural changes to the protocol. Alterations of this magnitude could not result in a protocol of the same name.

This is the end of the discussion about SSMP. The protocol cannot be adapted easily to perform the tasks required by the applications that should use it, so a new protocol must be designed to cope with the demands of these applications. This is the topic of the remainder of the report.
6.2 Customised terminal managers.

It should be clear by now that the most efficient protocol that could be implemented is one designed specifically for a given group of applications ([Macl83] and [HH82]). Such custom built terminal managers provide the maximum number of functions possible that perform exactly the tasks required by the application programs they are designed to support (almost optimal). However, these protocols may lack the capability to support additional applications, or may only support these in a very restricted way. Although general purpose protocols are preferable in most situations for their increased flexibility, customised protocols should not be ignored when an extreme improvement in efficiency is required for a small, fixed number of applications that are not expected to be modified in the near future. However, it is advantageous for a general purpose protocol be used where necessary to allow for the maximum possible flexibility to support applications that may be implemented unexpectedly in the future.
6.3 A Proposed Alternative.

An alternative protocol is required that is both flexible and efficient and that the host application can exert a great deal of control over. The alternative that is proposed here has been named the **Distributed Screen Management Protocol (DSMP)**

This protocol is based on SSMP but many of features of the protocol are very different from SSMP. The fundamental difference of the protocol is the method of processing used by the terminal manager. This is designed to reduce the processing required by the host computer to a minimum by processing some simple editing functions locally and transmitting only the result of this operation to the host. It is from this distributed processing that the name of the protocol has been derived.

The protocol is described in detail in appendix C, where most of the details of the protocol are described. The remainder of this section discusses the reasons behind the design decisions made for DSMP. Many of the features included in DSMP have resulted directly from the inadequate features of SSMP, however some have been included independently in an attempt to improve upon the overall efficiency of the protocol. The protocol provides as many of the features described in section 6.1 as possible to the best of its ability within the practical constraints of networking and processing facilities used. The major features of DSMP are as follows:

- A *host toolbox* of terminal interface functions has been provided to remove the need for applications programmers to handle minor protocol details.
- A generalised set of terminal manager *mode parameters*. These control many of the features that determine the flexibility of the terminal manager.
- The host program can *bind sequences of functions* to general conditions and evaluate these conditions in a host defined order
- Keystrokes are considered as *terminal events* and are assigned codes independently of the physical terminal codes.
- A simplified form of *session initialisation negotiation* is employed to make programming with DSMP simpler and to make session initialisation faster.
- *Data encoding* for command tokens is simpler and more efficient, *text compression* used on data strings, and fewer terminal tokens used.
- All *processing of lines* of text is done completely within the terminal manager. Only the result of the action is sent to the host program.
- A *macro facility* for defining and playing back keystroke macros.

**Host Toolbox.**

This is a library of functions that handle the details of communicating with the terminal manager. The major reasons for including the toolbox are:
6.5 Designing a Protocol

• A reduced workload on the programmer. The functions are very simple to use resulting in an increase in programmer productivity.

• A reduced number of transmission errors. The programmer is isolated from the data encoding and transmission details.

• Automatic processing of some input commands. The toolbox preprocesses all input data and filters the tokens passed to the host. Some of the tokens are processed by the toolbox to set global parameters for the host, while others are checked and passed to the host program for processing.

• Additional portability. Local alterations to the protocol may be hidden inside the toolbox functions. This allows the host program to be transported to another installation without modification, only a recompilation should be required.

Mode parameters.

DSMP has a set of terminal manager mode parameters that work on a similar principle to those of SSMP but the functions they provide are different. A full comparison of the functions available is given in appendix B. The functions offered that are different to SSMP are:

• A host definable escape keystroke to allow the actual escape key to assume another function.

• Built in cursor key-pad control and mouse tracking that can be switched on / off.

• Host definable action when the update limits are exceeded

• A mode where every user keystroke signals the host application, for "command" modes.

Binding Functions.

The method of binding terminal manager functions is very different to that used by SSMP. The manager builds a "tree" of conditions and corresponding actions. This has been done to provide a "programmable" host program interface where the application can specify exactly what it wishes the manager to do. The flexibility resulting from this scheme is huge since any number of primitive functions in a predefined sequence can be bound to a keystroke or any other event at the terminal that can be described with the predicate functions. The host application controls the binding and can build customised, high level functions by combining the primitives provided.

The tree of conditions is referred to as the "predicate tree". This tree has the structure of a linked list, but it is thought of as a tree because it is traversed both downwards to find a true predicate, and across to perform the associated consequent. The tree can be seen below.
Because the tree is built to contain only the functions that are required by the host, a minimal amount of searching is performed for each keystroke. Both building the tree initially and searching it after every terminal event requires a considerable amount of local processing, but since it is assumed that the local processor has sufficient capacity, this is not of concern. The functions provided by the terminal manager are summarised in appendix B.

**Terminal event representation.**

The translation of actual keystroke codes into standardised codes is a vital function for any terminal manager. An event based coding scheme results in applications software that is very portable and easily read and therefore encourages programmers to use the protocol. An event at a terminal is an action or condition as seen by the user, the best example of such an event is the entering of a keystroke by the user. The event is not described by the code the terminal key emitted, but by the label on the key that was pressed. Thus the problem is to find a method for representing an event (terminal interaction), as seen by the user, in a standardised format, so that all physical terminal characteristics can be translated into a standard representation. For the present, only keyboard interactions will be considered as events. With present computer technology, this does not restrict the definition, but in the future, new interactive devices may be introduced.

The first problem that is encountered when attempting to represent events is how to detect them. Obviously with keystrokes a code is sent to the terminal manager that can then translate the code into a standard representation. The problem comes when two different keystrokes send identical codes. A common example of this problem occurs when the 'tab' key was pressed followed by the command "ctrl/I". These keystrokes should be represented as independent, dissimilar events by the definition above, but the terminal manager receives identical codes from both of the keystrokes. This forces the terminal manager to the incorrect assumption that the two events are identical.

Detecting terminal events is very difficult if the terminal manager is not built into the physical terminal. Implementing the terminal manager in a concentrator or other device external to the
terminal itself results in a very restricted event detection scheme. The best place to implement a terminal manager is on a micro computer that is itself the terminal. This may allow the manager to detect individual keystrokes, and therefore produce an accurate representation of the events occurring at the keyboard. For brevity and efficiency, an integer representation of events was decided upon that allows every possible event to be statically bound to a unique integer. This mapping of events to integers must be an efficient operation because it is performed very frequently.

Even with a standard representation for every terminal event, problems still arise. When the host application initialises a session, it requests that specific terminal events be bound to certain terminal manager functions. The application could request a function to be bound to a event or keystroke that cannot occur at the physical terminal in question. It seems necessary, therefore, to have some form of negotiation to decide upon a set of events to use. This is the topic of the following discussion.

**Terminal initialisation negotiation.**

To efficiently initialise a terminal manager, a *minimal amount of negotiation* must occur. Ideally no negotiation would take place, which would result in the following benefits:

- Easier for application programmers to use
- Faster initialisation
- Less data being transmitted between the host and the terminal.

However, it is possible for an application to request an event that cannot be generated at the physical terminal to be bound to a function. In this case the request must be rejected. The question now is what should the host application do when it receives a rejection? If the solution to this problem is complex or requires the programmer to predict and handle hundreds of possible outcomes, the resulting protocol will quickly fall into disuse. Negotiation of some sort *must* take place, so to minimise the data transmitted and to provide an acceptably simple task for the programmer, the following scheme has been designed.

A database similar to the "termcaps" file in a UNIX™ operating system must be supplied. This file contains the *event codes and names* for all of the possible physical terminals that can be connected to the system (this will require constant maintenance). The event code is a standardised representation of a terminal event, while the event name is a short character string that describes the event (for example the *label* on a key). The event names can be used to build menus and other messages for display so that the host may refer to terminal events in a way that can be understood by the user. The format of the terminal event data base is as follows:
The toolbox of host routines contains functions to access this data base and retrieve various information required by applications programs. This allows a very fast and efficient decision to be made about the terminal events to be used. The routines provided allow events to be queried for their presence, and for associated names and codes to be found.

The event codes describe all of the possible keystrokes and other terminal events that can occur. The following keystrokes, however, do not correspond to terminal events when used alone. These keystrokes must be used together with another keystroke to produce an event:

- shift key
- control / option / alt keys
- "escape" key currently defined by the host

To initialise a session, the DSMP initialisation message is sent to the terminal manager in the same way as for SSMP. The terminal manager then returns a packet containing the initial mode parameter settings of the terminal including the identification number of the physical terminal that the application is now connected to. When the host receives these values, it sets its mode parameters to agree with those of the terminal and then accesses the terminal event data base entry for the terminal identification number received from the manager. This entry describes all of the events that can be received from the terminal. The host application can then scan this data to find a number of function keys, etc.. that it can use to perform the tasks it requires. These event codes then have terminal manager functions bound to them with a special command to the terminal. If an error is made and one of the event codes sent to the terminal is illegal then, if the terminal detects this, it can send a special command to the host application to inform the application that the session has failed. This failure message is handled automatically by the host toolbox which prints an error message and terminates execution. The terminal manager would already have reset itself to line mode.
6.9 Designing a Protocol

The events that are bound correctly to terminal functions are not acknowledged. It is assumed by the host application that if the terminal does not respond, the binding is valid. If an illegal binding is made that the terminal does not detect (it is believed that this may be possible), then the functions bound to that event will never be activated. Eventually the programmer will discover the "missing" function and reassign a new event to it.

A number of problems exist with this scheme, but these are mostly involved with organisational matters. These are:

- Inconsistencies can arise in the data bases between the host and the terminal
- All new terminals to a system must have a data base entry built for them
- Additional storage is required in the host computer
- The scheme requires some host computer processing at session initialisation. This is minimised by a wise initial choice of terminal events.

Data encoding.

The method of data encoding for network transmission used by DSMP is much simpler than that used by SSMP but it does result in a more compact data encoding. The primary design goal of the protocol was not to minimise the data transmission volume, but this was seen as highly desirable. The reason for the simplification in the encoding scheme was to reduce the processing required by the host computer. The coding scheme achieves a very high rate of information compression by employing some simple but effective algorithms. The coding scheme used is fully described in the reference guide, but the main ideas used are as follows:

- Only two special characters (command token, compression counter)
- No escape characters required, special chars reserved for one task.
- Simple one character representation for command tokens
- Only five command tokens that can be sent to the host to minimise decoding
- Strings of screen data treated as indivisible units for easy handling
- Text compression on all strings, multiple identical characters compacted
- Compact integer encoding (2 chars.) with error detection
- Special compact cursor position encoding (4 chars.)

Distributed processing.

The distributed processing involves the manager making all the changes to the "current line" locally, and then sending only the results of the editing to the host when the user moves to a new line. Thus the host program does not see any of the image editing functions, the only input it receives is as if the user had retyped the entire edited line. This allows the host program to be modified to allow it to accept strings of data very efficiently and to store these directly into its data structure. The Chef text editor was well suited to this scheme and it is thought that many other applications will have no trouble adapting to it. If an application cannot use this feature, it can specify that the terminal manager process in-line editing functions differently. When a new line is
added into the screen image, the host application is notified to perform the processing required to update the screen image. After this the local image editing treats the line the same as any other.

When a session is initialised, the local line editing buffer is configured to be compatible with the buffer used by the application. The application specifies the length of the edit buffer (up to some maximum). This allows the local editing to proceed exactly as if it were done by the host. The host can also specify the actions that will be taken when the edit buffer length is exceeded (truncation, error, etc..).

Although the scheme seems cumbersome at first, in practice it allows the input module of the host program to be substantially simplified and, by a series of "tricks", the data actually sent across the network is not as much as may be expected. These "tricks" include:

- Only sending the part of the line that has been changed. The line is sent from the first position changed to the end of the line.

- Not sending a line of data until it is absolutely necessary. A line is not sent to the host until some other line has been altered. This gives the user a chance to move away from a line and come back again without that line being sent.

A view of the current edit line within the Terminal Manager

![Diagram of edit line](image)

**Macro facility.**

A simple keystroke macro facility has been included to allow keystrokes to be recorded as they are processed. This operates by storing each user keystroke before its associated action is performed. When a playback is done, the stored data is feed back into the terminal manager's input as if it were being entered from the terminal. The definition and playback of macros is fully controlled by the host application which binds appropriate terminal events to these functions. This facility may not be compatible with many applications but it is hoped that by providing such a simple facility, these restrictions will be minimised.

**Other features.**

Many other minor features have also been implemented to improve the overall operation of the protocol. Some of these are described in the appendices C and D.
6.4 Implementing DSMP.

The experimental implementation of DSMP made for this report is very similar to the implementation of SSMP; a detailed description of the protocol may be found in appendix C. Both systems were implemented on the same computer with the same configuration of components. Again the Chef text editor was used, but this had to be modified again to accommodate the new protocol. A second module of Chef also had to have some alterations made to it also, in order for the editor to take full advantage of the local processing done by the terminal manager.

The terminal manager program for DSMP is slightly larger than for SSMP because of the additional processing required for the predicate tree. This can be seen in appendix G where a listing of the program is given. The host toolbox, as described in appendix D, is the same size because, although many more functions are provided than for SSMP, the functions require less host processing. The host toolbox and the altered parts of the host application may be found in appendices H and I respectively.
7. EVALUATION OF DSMP.

7.1 An Overview.

This section contains the results of an examination of the experimental implementation of DSMP built for this report. An evaluation of all of the features of DSMP was not attempted because Chef, the only host application used to test the protocol, did not support many of the features the protocol offered.

The major advantage the protocol offers over SSMP is the increased flexibility. DSMP is capable of supporting many more applications than SSMP due mainly to the method used to bind these events to functions. The disadvantages of DSMP are as follows, all of the factors listed have a larger effect than they did for SSMP.

- A (consistent) initialisation delay of about 1 - 2 seconds.
- A substantial processing requirement to initialise a session.
- A large demand on the local terminal processor
- High workload for the applications programmer
- Efficiency will decrease for applications that are incompatible with the local image processing functions of the terminal manager

Comparing DSMP to the ideal.

The ideal protocol discussed in section 6.1 includes some features that are very difficult to implement. A comparison of DSMP with the ideal features of a screen management protocol shows that DSMP supports many of the desired features. The features that are not supported are:

- Variable sized characters, only fixed sized characters can be handled.
- A generalised representation of white space
- Multiple virtual screens
- Multitasking and communication with multiple hosts.
- Compatibility with Postscript, etc..

These restrictions still allow compatibility with many more applications than SSMP. However, variable sized characters are used by many applications, so the failure of DSMP to provide these is now the major restriction of the protocol. DSMP requires that variable sized characters such as tabs are processed in the same way as suggested earlier for SSMP. Tab processing would not be too complex to add to the protocol, but many "rules" to control operation would have to be designed first.

DSMP is much closer than SSMP to the ideal screen management protocol, however, the protocol could be further improved with the addition of the features discussed above. The data
Evaluation of DSMP

channel management efficiency of the protocol is also better than that of SSMP and it is thought that little improvement on this scheme can be made.

**Users' and Programmers' view.**

The user of DSMP will notice no difference in "feel" for using SSMP. The protocols behave in a very similar fashion here because the delays for host processing depend on the commands entered and the speed of the host computer and network.

A programmer with DSMP does not have to handle the same number of possible situations as with SSMP, but a greater understanding of the protocol is required in order to "program" the terminal manager. It is expected that a good manual should solve most of the problems that could arise here, so overall the programmer workload is about the same.

**DSMP Distributed Processing.**

The major point for discussion about DSMP is whether the in-line functions should be completely processed locally by the terminal manager, or whether they should be processed in the same way as SSMP. The reason for using the distributed processing method was to minimise the workload on the host computer, while little attention was paid to achieving an absolute minimum amount of data sent. Even with these design criteria, the DSMP data encoding scheme is more efficient than the scheme used by SSMP. Thus, it was decided to retain the scheme because of the reduction in host workload and the availability of a generalised set of functions that could emulate the SSMP method of image editing if this was necessary.

The data sent across the network could be minimised by removing the distributed processing concept, but the initialisation data which makes up a significant part of the data for a short session would still be required. For this reason, and the low cost and high capacity of most data networks, the feature was included.

**Host processing.**

At session initialisation, a considerable amount of processing is required by the host computer to query the data base and calculate the terminal configuration parameters. After this, DSMP puts less load on the system than SSMP.
7.2 Measurements with Chef.

DSMP provided all of the features required by the Chef text editor. The editor could be implemented using DSMP so that its user interface behaved exactly the same as a directly connected version. As with SSMP though, it was still possible for the user to detect that the screen management protocol was in use by observing the delays for various commands.

The startup delay for DSMP is longer than for SSMP. This is not a problem as explained in section 5 because the user grows to expect the delay especially since the delay for a given application will always be the same. After screen mode had started, Chef was seen by the user as behaving in exactly the same way as the original version.

More of the Chef program had to be modified than for SSMP. Another module of the program was altered by removing the part that performed all the in-line functions. The same measurements were made for DSMP as for SSMP to gauge the value of the protocol. These results are shown in full in appendix E, a summary is given below:

- Average host wakeup improvement = 25
- Average data transmission improvement = 11

These terms were explained earlier. It can be seen from the results that the wakeup improvement of 25 is the same as for SSMP. This is expected since this generally depends on the number of events at the terminal that cannot be serviced by the terminal manager. The data transmission improvement, however, is better than that for SSMP, demonstrating that the data encoding scheme is superior.

DSMP also requires less host processing than SSMP, but this is very difficult to measure. An accurate method of measuring the amount of processing used for just the terminal interface module of Chef could not be found, and so no measurement was made. The claim above is made as a result of a comparison of the code that has to be executed by both implementations of Chef. This does not include the startup, only the operations during a session. The processing required by DSMP to initialise a session is greater than that required by SSMP.

Thus, for the Chef text editor, DSMP was much more capable of providing the services required than SSMP was. The improved generality of DSMP seems to be able to cope with many more situations than SSMP, including all of the functions of Chef.
8. SUMMARY.

The project has implemented and evaluated the Simple Screen Management Protocol (SSMP), and from the faults found with this protocol, another more generalised protocol, the Distributed Screen Management Protocol (DSMP), has been designed and implemented. Both of these protocols improve the efficiency of data transmission and host processing as well as provide an improved service to their user's. The major drawback of these protocols is that some functionality may be lost by the applications that use them. The major questions to be answered are:

- "Is a protocol really required ?"
- "Are the benefits from a SMP worth the effort of using them ?"
- "Are protocols like SSMP, or more complex protocols going to become widespread ?"

The results of the evaluation of the two protocols has demonstrated that a large increase in the efficiency of terminal communication results from the use of some protocol. Thus it is clear that there are benefits from using a screen management protocol (SMP) of some description to improve the service to users and reduce costs. Screen management protocols are required if a low cost, high quality service to remote users is to be provided. As with most modern software systems, an improvement in service to the users provides more complexity for the programmer. However, the requirements of the user are the most important consideration, so any improvement to the users interface will be worth a considerable programming effort. Other forces will also affect the outcome such as the low cost and high performance of micro processors. It may be thought that with a sufficiently powerful micro processor, connection to a host computer is not necessary. This is not true since electronic communications will always be necessary to communicate with distant users. Even if a "workstation" configuration is employed, users will notice a long delays across large networks due to propagation and switching delays. Thus it seems that screen management protocols will become quite common in the near future.

To provide a completely generalised facility, a much more complex protocol than SSMP or DSMP must be provided. A generalised protocol offers the benefit of portability which is of great importance for software designers and implementors. However, this must be a much more flexible protocol than SSMP to allow applications software to operate without restrictions. A protocol similar to DSMP seems to be what is required. It is not suggested here that DSMP should replace SSMP as many features of the protocol have to be refined and consolidated before it can be fully implemented.

Thus SSMP has a limited future. The protocol has many problems that must be corrected before even a simple editor such a "Chef" can operate correctly when using it. It is thought that a protocol similar to DSMP will play a dominant part in finding the best way for remote users to communicate across data networks.
9.1 Appendices

9. APPENDICES.

Appendices F to J can be found in the accompanying line-flow folder.
Appendix A:

References.

[SSMP85] J. A. Hunter
(sponsored by the U.K. Joint Network Team)
The Simple Screen Management Protocol
Computing Laboratory,
University of Newcastle upon Tyne, England.
July 1985

[DIX85] W. G. Dixon
The SSMP Terminal Emulator for the BBC microcomputer
University of Cambridge computing service,
November 1985

[HH82] J.A. Hunter and N.F. Hall
A Network Screen Editor Implementation
Software Practice and Experience
Vol. 12, no. 9
1982

[Macl83] M. A. Maclean
"A terminal handler model for a display editor" &
"The Chef display mode terminal handler"
private communication
University of Canterbury
February 1983

[Macl82] M. A. Maclean
"Network terminal for the Chef editor" &
"Chef terminal concentrator protocol"
private communication
University of Canterbury
November 1982
Appendix B:  
A Summary of Terminal Manager Functions.

This appendix shows all of the terminal manager functions available for both of the protocols SSMP and DSMP. The functions provided by DSMP perform more "primitive" types of operations than those of SSMP. This gives DSMP a great deal more flexibility because it allows host applications to combine these functions to make up complex, customised procedures that efficiently perform the tasks required by the host.

<table>
<thead>
<tr>
<th>SSMP functions</th>
<th>DSMP functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Session control functions</strong></td>
<td></td>
</tr>
<tr>
<td>interrupt host</td>
<td>emergency exit (panic)</td>
</tr>
<tr>
<td>suspend session</td>
<td></td>
</tr>
<tr>
<td>restart session</td>
<td></td>
</tr>
<tr>
<td><strong>Cursor movements</strong></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>move to coordinate (x, y)</td>
</tr>
<tr>
<td>cursor arrow movements</td>
<td>cursor arrow movements</td>
</tr>
<tr>
<td>move to next / previous tab</td>
<td>move to next / previous tab</td>
</tr>
<tr>
<td>move to first non space</td>
<td>move to next / prev. word</td>
</tr>
<tr>
<td>move to after last non space</td>
<td>move to first non space</td>
</tr>
<tr>
<td>move to new line</td>
<td>move to after last non space</td>
</tr>
<tr>
<td>move to left of window</td>
<td>move to auto wrapping position</td>
</tr>
<tr>
<td>goto home window</td>
<td>move to new line</td>
</tr>
<tr>
<td>goto next / prev. window</td>
<td>move to all window bounds</td>
</tr>
<tr>
<td>In-line image editing functions</td>
<td>goto a specified window</td>
</tr>
<tr>
<td>-</td>
<td>goto next / prev. window</td>
</tr>
<tr>
<td>erase previous character</td>
<td>store the current character</td>
</tr>
<tr>
<td>insert a space</td>
<td>erase prev. character</td>
</tr>
<tr>
<td>delete a character</td>
<td>insert a space</td>
</tr>
<tr>
<td>erase to end of line</td>
<td>delete a character</td>
</tr>
<tr>
<td>erase a line</td>
<td>erase to end of line</td>
</tr>
<tr>
<td>Inter-line image editing functions</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td></td>
</tr>
<tr>
<td>insert a line</td>
<td></td>
</tr>
</tbody>
</table>
delete a line
append a line after current line
split the current line

Other functions.
insert mode on/off
- toggle any mode element
- set any mode
- toggle a tab setting
- repaint screen
- begin / end macro definition
- replay a macro.

Terminal manager parameters.
dinvalid
notify
selectgr
txtface - video attributes of text
reqshift
icharmode
ichmode - char. insert mode on / off
ilinerow
- bounds - action at window bounds
intsignal
killkey - host defined panic key
esckey - host defined esc key
cursor - cursor pad on / off
- ksigmod - signal every key on / off
Appendix C:

Technical Details of DSMP.

I Introduction.

The Distributed Screen Management Protocol (DSMP) provides a large number of terminal screen functions that can be used by a wide variety of host applications to manage remote terminals in a generalised fashion independently of individual physical terminal characteristics. The protocol provides a "programmable" type of interface to the host program so that as many functions as possible may be implemented exactly without any modification to the operation of the host application program. This programmability is provided through a special binding function that operates on a large number of primitive functions to build up powerful commands tailored exactly to the host program's needs.

The actual internal operation of the protocol can achieve large savings in the data transmitted between the terminal and the host computer. Savings can also be made in the number of host "wake-ups" or terminal interactions, and especially in the amount of processing done by the host computer.

A DSMP system consists of two components, the terminal manager and the host application. The terminal manager is a program that processes user terminal interactions and host screen manipulations from a computer close to the physical terminal. The host application is a program that runs within a distant host computer. This program uses the host toolbox which is a library of functions that allow the host to communicate easily across a data network with the terminal manager.

This guide describes the protocol and how it is used. Many of the features of this protocol are similar to SSMP and for this reason they are not explained in great detail. For a full description of SSMP, the reference [SSMP85] should be consulted.
.2 Shared Data Structure.

The "shared data structure" is a small group of parameters that are maintained by both the terminal manager and the host toolbox so that either can easily query their common "settings" in an efficient manner. The shared data structure describes the state of a session at any given time by storing the following values:

- access token
- row, col
- mode parameters

The "access token" controls the transmission and processing of the two components of the system. The terminal manager can only transmit data and process user keystrokes while it has the access token, and similarly, the host can only transmit and update the shared data structure while it has the access token. This half duplex operation is vital to the successful operation of the protocol, if ever the token is duplicated or lost, the protocol will fail. The host programmer must remember to pass the token back to the terminal as soon as possible at all times. The terminal will always wait until it receives the token before it continues processing user keystrokes, so if the token is not returned, a deadlock will occur. While the host has the token, the terminal manager buffers all of the user's keystrokes for processing when it regains the token.

The "row" and "col" variables indicate the current cursor position. These are automatically maintained by the toolbox so that whenever the cursor position is required, these variables can be read directly. The row and col variables should never be assigned to directly, an attempt to do this may have destructive results. The current cursor position is the most important and most frequently used information in the protocol because it always influences the action taken by the application. It should be remembered that the cursor position is never written to the output buffer directly. Only when "the next" function is output is the cursor position output before it. This allows the cursor to be moved many times but for only the final cursor position to be transmitted as a result.

The "mode parameters" describe terminal settings that have been specified by the application or the user. This data is shared so that the host program may query the state of a terminal parameter at any time. The parameters here are changed by calling the toolbox function "tp_setmode" that changes the host's copy of the appropriate parameter and then informs the terminal to change its copy. This function operates as follows:

```c
tp_setmode(idx, val)
```

where "idx" is the name of a parameter that is to be changed, and "val" is the new value of the parameter. The values of the parameters may be queried in the following way:

```c
what_val = tp_query(idx)
```

Also, the user can change the terminal's copy of some parameters at any time (depending on the functions provided by the host). The toolbox automatically updates the host's copy of the parameters when the terminal notifies the host of this change.
The parameters can be divided into two categories. The first of these contains a group of parameters that can only be changed by the terminal manager. Attempts to alter their value from the host application using the toolbox functions will fail. These are designed as read only parameters for the host program to determine what state the terminal is in. The second category contains parameters that can be changed by both the host and the user. The mode parameters are:

**Read only parameters.**
- ttermid - the identification number of the physical terminal
- tmmaxrow - the max. screen row
- tmmaxcol - the max. screen column
- tmacnum - the number of the macro currently being defined
- tmmaxwin - the maximum number of windows specified by the host
- tcurwin - the currently active window

**Read / Write parameters.**
- killkey - the kill key code (panic button)
- esckey - the escape key code
- cursor - zero = the cursor key-pad is active, one = deactivate the cursor key-pad.
- bounds - the action when the window bounds are exceeded, values are:
  - bxsigl - signal the host
  - bxbell - ring the bell and ignore
  - bxignr - ignore
- ksigmod - zero = normal operation, one = host signalled for every keystroke
- ichmode - zero = char insertion mode off, one = char. insertion on.
- txtface - the video attributes of text to be displayed, values supported:
  - sgr_p - plain text
  - sgr_u - underlined text
  - sgr_i - inverse text (reverse video)
  - sgr_c - concealed text (invisible)
- rectupd - zero = cursor can move anywhere in window, one = cursor can't exceed the end of line for each line of text.

All of the constant names and their values are defined in the DSMP header file (hdef.h). This file should be consulted before an implementation is attempted to check if modifications have been made to the local version of DSMP.
3 Screen Management Sessions.

This is the time between a successful DSMP initialisation and a successful shutdown. A host program must initialise the DSMP toolbox before any terminal manager communication can be attempted. Once initialised, the toolbox is ready to be used for any number of screen management sessions. An algorithm used by many programs to communicate with the terminal manager is as follows. This algorithm shows how to setup, use and shutdown a DSMP session.

host_session_algorithm:

begin
    tp_setup();
    while (not end_of_program) begin
        if (not tp_session(0))
            error("terminal not responding");
        forall (modes)
            tp_setmode(idx, val);
        forall (actions)
            tp_setfunc(segn, "predicate | consequent");
        forall (windows) begin
            tp_window(l, w);
            tp_update(window parameters);
        end;
        while (not session_end_key) begin
            {**** host program interface goes here ****}
        end;
        tp_session(1);
    end;
    tp_shutdown();
end;

These sessions cannot be nested, each session initialisation command, except for the first, must have been preceded by a session shutdown command. The reason for this minor restriction is the simplification of the protocol and the improved control and error detection that results. During a session initialisation, the terminal identification number and the screen dimensions are set by the host toolbox from a special initialisation packet from the terminal manager. This allows the host to access the terminal events data base (described later) and to query the maximum row and column numbers of the screen. The numbering of rows and columns begins at zero.

The protocol provides two methods for exiting from a screen management session. One of these methods as demonstrated above (tp_shutdown), is the normal way to exit from a session.
This is done after a user has entered an "exit" key specific to the application. The host application then tidies up the terminal screen image and instructs the terminal manager to exit from screen management mode. The other way to exit a session is to use the special emergency shutdown method for recovering from system errors. This "panic button" is a special host application defined keystroke, (the default is "esc, ctrl/X"), that exits the terminal manager from screen management mode immediately, then informs the host of this action. If the host program is still operative, it may respond to this request as defined by the applications programmer to return the user to a "safe" state of his choice. If the host program has failed, the user's terminal will be returned to "line mode" and will not be locked awaiting a host response. The panic button is not queued in the keystroke queue by the manager at any time, it always acts immediately and causes the return of a special token to the host.

It can be seen that the terminal manager does not operate completely transparently. The user is required to be aware of the presence of the terminal manager and to know the panic button sequence. It is considered to be impossible to have the terminal manager operate completely transparently to the user, but the single panic button is the simplest mechanism possible that even the most inexperienced user should be able to use. This method of error recovery relies very heavily on the implementor clearly publicising the panic button for every application. It is highly recommended to make this button the same for all applications so the user community can easily remember the sequence.

**Terminal events.**

The characteristics of the physical terminal are isolated from the host application. It is necessary for the host application to know only what terminal events are possible for a given physical terminal. This is determined from the terminal identification number (ttermid) set by the terminal manager at session initialisation time. This mode parameter is an integer value that uniquely identifies the terminal connected to the host for a given screen management session.

The host application may use this identification number to access a data base of terminal events to find a suitable set of events to bind to the required terminal manager functions. The format of the data base is as follows:

```
"term id." = "event code":"event name"; "event code"; "..."
```

This allows a full list of the events possible at a given terminal to be found, so that the host application may consult this list and make a decision about which events it will use, without the need for negotiation with the terminal manager. This eliminates many of the problems and inefficiencies of negotiation, especially since the programmer can survey the file to find "good" events to use before building an application. This can save some negotiation with the data base at a session initialisation.

The event names are stored in the data base so that the terminal events may be referred to in menus and other display devices in a user friendly way. These names are character strings up to
six characters long and are usually the names of labels on keys that generate events.

For every new terminal introduced to a DSMP system, a new entry in the data base of terminal events must be made and the terminal manager must be informed. The entries in the data base must always be accurate, so constant maintenance is required to keep this data base up to date.

To allow the user to enter a wider range of keystrokes, the following keys do NOT produce terminal events when used alone. These keystrokes must be followed by one other keystroke in order to produce a terminal event. The keys are:

- shift key, caps lock
- control, convert function
- alternate, option
- escape (as currently defined by the host application)

There is no corresponding entry in the terminal event data base for these keystrokes, however, there are event codes stored for sequences such as ctrl/A, option/I, etc.. The escape key that does not have an event code representation is the escape event currently defined by the host application. This is not necessarily the physical escape key. This scheme is implemented by storing an event code for the escape key itself, and then using the toolbox routines that access the data base to "filter out" the current escape key and make sure that it cannot be bound to a function.
.4 Host Toolbox Functions.

The Host toolbox functions are described separately in appendix D which provides a "programmer's guide to the toolbox" with a brief description of all the functions available in the toolbox. These functions either communicate simple commands to the terminal manager using the DSMP encoding scheme, and accept command tokens from the manager, or query data objects such as the shared data structure, or the terminal event data base. The functions also maintain the input and output buffers, flushing or filling them at the appropriate times.

The toolbox should be used by all applications programs to communicate with the terminal manager. All the functions available to control the terminal's operation are provided in this toolbox which has many functions similar to those described in the Fawn book [SSMP85]. The more important of these functions are discussed below:

Binding user functions.

DSMP, unlike SSMP, does not bind functions explicitly to specific user entered keystrokes. DSMP binds a series of functions, to be performed in the given order, to an abstract condition. This condition must be a series of the standard predicate functions provided. The binding of predicates to consequent functions is done with the toolbox function "tp_setfunc". This is one of the most complex of the toolbox functions since a great deal of error checking and code generation is performed by this function. Each predicate function is assigned a sequence number that determines the order of evaluation of the predicates by the terminal manager. The sequence number (seqn) must be a unique integer, and is used to force the order of evaluation in the predicate tree (discussed later). The format of the tp_setfunc command is as follows:

\[
\text{tp_setfunc(seqn, "predicate | consequent");}
\]

The predicate is a series of predicate functions and the consequent is a series of consequent functions. This predicate string is parsed and the resulting data is sent to the terminal manager. Any errors detected during the parsing causes an error message to be output to a file and the binding request to be ignored by tp_setfunc. This is the only safe way to handle errors here because a screen management session will be in progress, and to take any other action may have an unpredictable effect. A function can be disabled or cancelled, with the following predicate string.

\[
\text{tp_setfunc(seqn, "}")}
\]

The predicate string.

This section contains a description of the language for specifying function bindings in the "tp_setfunc" toolbox function. Each instance of a string of this language is known as a predicate string. The following description uses a form of the BNF notation, literal symbols are displayed
in boldface.

\[
\text{pred. string} ::= \text{" predicate | consequent "}
\]
\[
\text{predicate} ::= \text{predicate function list}
\]
\[
\text{consequent} ::= \text{consequent function list}
\]
\[
\text{function list} ::= \text{function list ; function spec | function spec}
\]
\[
\text{function spec} ::= \text{function def, param list | function def}
\]
\[
\text{function def} ::= \text{function name | not, function name}
\]
\[
\text{param list} ::= \text{param list, constant | constant}
\]
\[
\text{function name} == \{\text{any valid function name in context (below)}\}
\]
\[
\text{constant} == \{\text{any legal integer value in context}\}
\]

Some examples of predicate strings that instruct the terminal manager to bind a keystroke to the delete character function and to detect and process an automatic word wrapping are as follows.

\[
\text{tp_setfunc}(3, \text{"keq,153 | dc"});
\]
\[
\text{tp_setfunc}(42, \text{"cin,37,200; kpr; not, keq,32 | ch; lw 37; xt,177; el"});
\]

The first example performs the delete char. (dc) function whenever an event (keystroke) with code 153 occurs. The second performs the consequent when the cursor is between columns 37 and 200, the last keystroke was of a printable character and the character was not a blank (ASCII 32). The consequent stores the character, moves to the line wrapping position (wrap col = 37), sends the event code 177 to the host to indicate that an auto wrap has occurred, and erases to the end of the current line from the cursor position. Functions can be performed after the token has been passed back, and in fact this produces a more efficient operation. The terminal manager will process all of the functions in the consequent list before it accepts any input from the host. Thus the terminal manager still has exclusive rights to the screen image even though the token has been passed to the host.

The predicate and consequent functions provided in the experimental implementation (version 1) of the protocol are as follows:

<table>
<thead>
<tr>
<th>Predicate functions</th>
<th>mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>key equal to ( n )</td>
<td>KEQ</td>
</tr>
<tr>
<td>key printable</td>
<td>KPR</td>
</tr>
<tr>
<td>row in range ( r_1, r_2 )</td>
<td>RIN</td>
</tr>
<tr>
<td>col. in range ( c_1, c_2 )</td>
<td>CIN</td>
</tr>
<tr>
<td>cursor in box ( r_1, c_1, r_2, c_2 )</td>
<td>BOX</td>
</tr>
<tr>
<td>Consequent functions</td>
<td>mnemonic</td>
</tr>
<tr>
<td>----------------------</td>
<td>----------</td>
</tr>
<tr>
<td><strong>Cursor movement functions</strong></td>
<td></td>
</tr>
<tr>
<td>move to coordinate ((r, c))</td>
<td>SC</td>
</tr>
<tr>
<td>cursor arrow movements</td>
<td>KL, KR, KU, KD</td>
</tr>
<tr>
<td>move to next / previous tab</td>
<td>NT, PT</td>
</tr>
<tr>
<td>move to next / prev. word</td>
<td>NW, PW</td>
</tr>
<tr>
<td>move to first non space</td>
<td>LL</td>
</tr>
<tr>
<td>move to after last non space</td>
<td>LR</td>
</tr>
<tr>
<td>move to auto wrapping col.</td>
<td>LW</td>
</tr>
<tr>
<td>move to new line</td>
<td>NL</td>
</tr>
<tr>
<td>move to all window bounds</td>
<td>WL, WR, WT, WB</td>
</tr>
<tr>
<td>goto a specified window ((n))</td>
<td>WG</td>
</tr>
<tr>
<td>goto next / prevWINDOW</td>
<td>WN, WP</td>
</tr>
<tr>
<td><strong>Image editing functions</strong></td>
<td></td>
</tr>
<tr>
<td>store char. at the cursor position</td>
<td>CH</td>
</tr>
<tr>
<td>erase previous character</td>
<td>BS</td>
</tr>
<tr>
<td>insert a space</td>
<td>IC</td>
</tr>
<tr>
<td>delete a character</td>
<td>DC</td>
</tr>
<tr>
<td>erase to end of line</td>
<td>EL</td>
</tr>
<tr>
<td><strong>Other functions</strong></td>
<td></td>
</tr>
<tr>
<td>return token with a code ((\text{code}))</td>
<td>XT</td>
</tr>
<tr>
<td>return token with current key</td>
<td>XK</td>
</tr>
<tr>
<td>toggle any mode element ((n))</td>
<td>TM</td>
</tr>
<tr>
<td>set any mode ((n))</td>
<td>SM</td>
</tr>
<tr>
<td>toggle a tab setting ((c))</td>
<td>TT</td>
</tr>
<tr>
<td>repaint screen {all windows}</td>
<td>RS</td>
</tr>
<tr>
<td>begin / end macro definition ((n))</td>
<td>MB, ME</td>
</tr>
<tr>
<td>replay a macro ((n))</td>
<td>MR</td>
</tr>
</tbody>
</table>

**Accessing the terminal data base.**

The terminal event data base stores information required to bind terminal manager functions to terminal events. This data base is accessed with the toolbox functions "tp_eventp", "tp_eventname", and "tp_eventcode". The function "tp_eventp" returns 1 if its parameter is in the data base under the entry for the current physical terminal identifier, otherwise it returns 0. The correct data base entry is selected automatically by the toolbox session routine when it receives a valid terminal identifier from the terminal manager. Thus the only way to access the terminal data base is via the toolbox functions.

The "tp_eventname" function returns the name of the event identifier it is passed. For
consistency the reverse operation "tp_eventcode" is also provided. It is not recommended that this 
function be used a great deal. It is very inefficient because it must perform pattern matching on 
character strings. Examples of the use of these functions are as follows:

```plaintext
if (eventp(123)) then aname := tp_eventname(123)
else acode := tp_eventcode("insln");
```

**Input from the terminal.**

The toolbox function "tp_get_screen" is used to receive all input to the host application of these 
command tokens. This function automatically processes the tokens that set the shared data 
structure and error tokens. The actions it performs are:

- sets cursor position is shared data structure
- sets mode parameters
- sets the current window
- terminates the host application with error messages

The remaining tokens that arrive are passed back to the host program for processing. Thus this 
function works as an input decoder and filter. All terminal events that cannot be processed by the 
terminal manager are passed to the host application as parameters to access token commands, where 
the parameter value indicates the event that occurred. When a host program gets a token command, 
it should look at its parameter with "tp_get_int" to see what event occurred. A value of zero 
indicates that no specific event occurred, so the token can be passed back with "tp_token(0)".

The basic input algorithm for the host application should be:

```plaintext
xtok := tp_get_screen(0);
if (is_access_token(xtok)) then
  case (tp_get_int()) of
  0 : tp_token(0);
  123: insert_a_line();
    . . .
  42 : tidy_up();
  end; {case}
else if (is_exit_token(xtok)) then
  tp_shutdown();
  fast_stop();
end;
else
```
5 The Terminal Manager.

The DSMP terminal manager may be implemented within a terminal concentrator but, to attain the maximum performance from the protocol, it is recommended that the terminal manager be implemented within a dedicated micro computer. Actions such as mouse tracking will not lag behind the user due to time slicing or interrupt delays if a dedicated processor is used. It should be remembered that DSMP has been designed to minimise the workload on the host computer, and that this involves maximising the workload on the local processor.

Terminal Manager Functions.

The functions provided by the DSMP terminal manager are simple, indivisible functions (primitives) that can be combined to produce large and complex operations specific to the host program that requested them. A list of the processing (consequent) functions provided by the manager is shown above in the description of the toolbox function "tp_setfunc". These are all of the functions provided by the terminal manager, and the only way to bind these functions to events is with the "tp_setfunc" function.

The "cursor movement" functions have no effect on the screen image data and write nothing to the output buffer. These functions just reposition the cursor and set a flag to indicate that the cursor has been moved. The next time a command token is output, and the cursor-moved flag is set, the new cursor position is placed immediately before the token in the output buffer.

The "image editing" functions alter the screen image in some way and again do not write any data to the output buffer. The terminal manager has a concept of a "current edit line" which is the line of the screen image that the cursor is currently positioned on. When an image editing function is invoked, the terminal manager alters the contents of the current edit line. Whenever the user modifies another line, and the current edit line has been altered, the edit line is written to the output buffer, and the new line becomes the current edit line.

The process of writing the edit line to the output buffer is simple, but some "tricks" are employed to reduce the amount of data that is transmitted. The position of the first character in the line that was altered is stored by the terminal manager. Thus the line is only output from this column to the "end of line", and is preceded by the cursor position corresponding to the position of the first alteration. Also, when all strings are written to the output buffer text compression is used. The data transmission considerations are discussed in the following chapter. When the host receives this data, it can take the same action as if the user had moved the cursor to the indicated position and retyped the line of data. This allows an efficient host interface since only one type of image editing command is ever expected.

The "other" functions (listed in the host toolbox section) provided by the terminal manager perform miscellaneous group of tasks, these are described here very briefly. The macro functions allow macros to be defined and played back by the user, repaint allows the user to repaint the physical screen image when this is distorted in some way, and the tab function allows the tab stop
in the current column to be toggled on and off. This allows users to redefine their own tab positions. The remaining functions allow the user to change the state of the shared data structure. The first of these simply returns the access token to the host with either the current key code or a host specified key code, and the second allows the mode parameters to be changed. This toggles or sets a host specified mode parameter to a host specified value.

**Function binding.**

The terminal manager functions must be bound to events that may occur at the terminal before these functions can be invoked. The binding of these functions has been described above in the host toolbox section where a guide to how to make a binding was given. This section describes the way in which the terminal manager processes the binding information and how the event scanner works.

When a "setfunc" token arrives in the input buffer, the following data is built into a list, known as the *predicate tree*. It is called a tree because of the way in which it works and not the way it is stored. This tree is built up as these tokens arrive at the terminal. Each token causes a new node to be inserted into the tree at a position governed by the sequence number of the new predicate, the tree is always ordered numerically by predicate sequence numbers. New nodes may be added and old ones removed at any time during a screen management session.

When an event occurs at the terminal, this is nearly always the user entering a keystroke, the terminal manager first scans the predicate tree for a true condition. The ordering of the tree allows complex conditions to be constructed by the host program (logical "or" operations). If a true condition is found, the corresponding consequent (action) for that predicate is performed and the next event awaited. If no predicate evaluates to true, then a default action is performed (this saves time during session initialisation). The defaults can be short circuited by specifying a condition that is equivalent to them that perform some other actions. Since these new conditions will be stored in the tree, they will be evaluated before the defaults. The default actions are:

- kpr | ch • if key is printable, store it in the screen image
- keq "del" | bs • if the "del" key was entered, erase the previous char.
- otherwise | xk • all other events cause the event code to return to the host

This continual scanning of the predicate tree may appear to be very inefficient, but the tree is not usually very long and the local processor is assumed to have sufficient capacity.

**Keystroke translation.**

The events triggered by the user at the terminal must all be assigned an integer code. The only events that can occur at present are user entered keystrokes. However, in the future, this may be extended to other input devices. An event is, therefore, an action as *seen by the user*. Thus the codes that are associated with keystrokes must be associated with the *label on the key that is pressed* (as seen by the user) and must be independent of the code generated by the keystroke.
The standardisation of codes to represent events at the terminal is one of the most important functions of the terminal manager. If no standardisation was provided, the host would have to communicate differently with each physical terminal to which it could be connected. This would quickly become very complex and difficult to manage.

The event codes that are generated are not discussed in detail here, but these would be simple integers that are calculated easily from the codes sent by the keystrokes. Such a scheme would represent the printable characters as the integer equivalent of their ASCII values, and special function keys would be numbered arbitrarily in some logical order.
.6 Data Encoding and Network Transmission.

The major features of the DSMP data encoding scheme are:

- There are only two special characters (ctrl chars.)
- Only printable ASCII characters are used to encode the data
- Command tokens are encoded as a single character
- Strings may be sent as indivisible units
- Integers are encoded in a very compact format (2 chars.)
- A special compact encoding for the cursor position is used (4 chars.)
- Text compression is used on string data for blocks of identical chars.
- There is a very small number of terminal tokens

The data transmitted between the host and the terminal is composed of five types of object. All of the characters transmitted are either printable ASCII characters or the characters 'G' or 'K'. This has been done to minimise the possibility of control characters disturbing the underlying network software. The choice of the two special characters has been made carefully to try to avoid undesirable effects on sensitive networks. The five objects are:

- token symbols ('G - bel)
- repeat symbols ('K - vt)
- command tokens (a character - see below)
- token parameters (integers)
- character strings

The "token symbol" is a special character used to denote a command token by being placed in a data packet before every command token. This character is unique (ie. it cannot appear anywhere else). The "repeat symbol" is used for text compression to indicate that the next character is to be interpreted as an integer value which represents the number of times the character following that is repeated. Thus 10 'x' characters would be represented as "^K&x" where '&' is the character that represents the value 10.

The "command tokens" used for communication between the terminal and the host are given below. These represent all the functions that can be signalled by the terminal. All of the following command tokens are represented by a single character in their encoded form. The "set" tokens below ("T_S..") are used to update the hosts copy of the shared data structure to the same state as the terminal's copy.

**Tokens from the Terminal.**

T_TK (x) - passes the access token to the host with event code x
T_SC (r, c) - sets the cursor position
T_SM (i, v) - sets the mode parameter i to value v
There is also a much larger set of tokens that are sent from the host to the terminal. These correspond almost exactly to each of the toolbox functions and therefore are not discussed here. If interested, these can be found in the DSMP header file (hdef.h).

**Token parameters** are integer values that follow command tokens in the data stream. The coding scheme used for integers translates their value into a two character representation. Thus any integer in the range between one and 800 can be represented by two characters, and zero is represented by a single character. All integers outside this range are illegal. This very compact scheme requires surprisingly little processing to encode and decode. It is a minor saving to represent zero as a single character since it is encoded much more frequently than any other value.

The final objects in a data transmission are character strings. These are simple sequences of characters that may represent any data is to be sent through the channel. These strings are always subject to the text compression algorithm described above.

A special encoding is used for the "set-cursor" command to produce a highly compacted instruction for this very common command. The encoding used represents the values of the row and column as single characters (0 .. 80), and uses two more characters (the token symbol and token code) to complete the instruction.
.7 Version one exceptions.

In version one of DSMP the toolbox does not provide the sophisticated function binding language described in this guide. It instead supplies only a crude interface with a variable number of parameters that specify the binding in a very similar way to that described here. For details of the exact method used, the source code should be examined, however, this should not be a permanent feature. The functionality of this arrangement is identical to that described above, only the interface is different.

The terminal event data base has not been implemented. Version one was designed to communicate with only one type of terminal. It was therefore a waste of time to incorporate a data base for this. The terminal characteristics must be "hard-wired" into any applications that wish to use this implementation.

Macros are also not supported in version one. The terminal manager has all of the mechanisms for using macros, but an operational implementation of these was never made. This was because version one is an experimental version, and macros added nothing to the experimental measurements that could be made.

For easy testing, the special data transmission symbols have been redefined as printable characters. This allows the data sent between the host and the terminal to be examined without the problems associated with printing control characters to output devices. The symbols used were: token symbol = '~', repeat symbol = '|'. These symbols cannot be used as data otherwise the protocol will crash! An example of some data encoded with this scheme can be seen in appendix F.
Appendix D:

DSMP Host Toolbox.

This appendix contains a brief description of the operation of each of the toolbox functions supplied by the DSMP system. All of the functions in the toolbox are prefixed with the string "tp_" (terminal protocol) to clearly identify that they are DSMP toolbox functions.

(Note: All parameters to the following functions are of the type "int" in the "C" programming language unless otherwise specified.)

Toolbox files.

hdef.h

This file should be included in the host application program terminal interface modules. The header file contains the control token, and other definitions and a number of other constants required by the DSMP system. The file should be inspected before an implementation is attempted so as to familiarise the programmer with all of the facilities and restrictions of the system.

Load modules:

htools.o - contains the toolbox functions

dqueue.o - support to the htools module.

dpipes.o - support for the dqueue module.

These files make up the terminal manager communication toolbox. They contain the functions discussed above and should be linked with any Host Application program that wishes to use the DSMP protocol. All input and output to and from a terminal must be made using these functions once a DSMP session has been established.

The modules contain all of the functions required by the host in a layered structure. Every site that has DSMP available should supply the source code for these modules so the programmer can inspect this if required to determine specific details of the operation of the system. This code should not be able to be modified by unauthorised personnel.

The "htools" module contains the high level functions for terminal input and output. These are all the functions that a host program should ever require. The other modules are supplied to provide support to the these functions by performing low level or primitive operations. If the host program needs to perform a very low level task with the terminal or to escape the structure of DSMP, a VERY experienced programmer may need to use the low level functions. This is NOT recommended.
Input from the Terminal.

**tp_get_screen(npkt)**

This is the major input function. It accepts packets from the terminal manager, decodes them, and then either performs an action required by them or passes the tokens back to the host application program. There is only a small set of command tokens that can be sent by the terminal manager. The simple tokens such as "set-cursor", "set-mode", and "set-window" just set the appropriate global variables in the DSMP header file. The other tokens such as "access-token", "character-string", and "shutdown" cause the token to be passed back to the host for processing. If the parameter of the shutdown function is not zero, this function will print an appropriate error message and terminate the host application. An error of this type can't be recovered from, so a termination is the only remedy. A zero parameter causes the token to be passed back for processing, so the parameter should not be read by the application. All illegal token as skipped until a recognisable token is found.

The function takes one parameter that specifies the maximum number of packets that it is to accept before returning control to the host program. This occurs in a "packet mode" that involves waiting until a packet arrives, acting upon the contents of the packet and then waiting for the next packet. If ever a token that cannot be processed locally is encountered, the token is returned immediately to the host program. If "npkt" is set to zero, an indefinite number of packets will be read before the next token is returned to the host.

**tp_get_char( )**

This function is used for reading character strings from the input. The character is returned that appears next in the input buffer if the last command read from the terminal manager with tp_get_screen was a "character-string" command. If the last command was not a character command or if the end of the current string is encountered, (ie. a command is found), this function returns zero. Thus it is possible to efficiently read a string from the input buffer once a string has been found.

If a string command is read, and the tp_get_char function is not called to read the string that follows, the string is skipped when the next command is read. To copy a string to a variable in the host, the following loop construction should be used:

```c
if (tp_get_screen(0) == P_CH)
    while (ch = tp_get_char()) localVar[spos++] = ch;
```

The packetisation of data may mean that a string is split across a number of packets, so when the end of a packet is encountered, zero is returned. This may cause some minor problems, but the continuation of the string in the next packet begins with another "character-string" token. This
apparent very low level method is necessary to provide sufficient flexibility to all that wish to use the system.

\textbf{tp\_get\_int( )}

This function reads an integer from the input buffer. It checks for an empty input buffer and refills it if necessary before reading the required integer. This may involve a time sleeping while waiting for a packet. This function is used most commonly to get the argument of the access token. The integer returned is decoded from the next data object found in the input buffer. If the next object found is a command or the end of the packet, zero is returned.
Output to the terminal.

`tp_alarm( )`
Puts a token in the current packet to instruct the terminal manager to sound the bell on the terminal.

`tp_clrscreen( )`
Cleans the entire terminal screen. Only the physical screen is cleared, not the data stored in the window data structure.

`tp_clrwindow( )`
Clears the window data structures to null in the terminal manager. The manager stores all the data displayed in a window in a special data structure. To erase the contents of a window, this function should be called to remove the stored data and the "tp_display" function should be used to update the physical screen image of the window.

`tp_delchar(n)`
Deletes the previous n characters from the current cursor position. If n equals one, the character at the current cursor position is replaced with a blank character.

`tp_display( )`
Displays the current active window on the physical terminal screen. This function is used to redisplay the screen when the host believes that the screen image no longer reflects the contents of the window data structure.

`tp_eraseteln( )`
Replaces all characters from the current cursor position to the end of the current line with null characters, i.e., erases to the end of the current line. The cursor position is unchanged.

`tp_flush_pkt( )`
This function flushes the current packet. The token is NOT passed back to the terminal, only the current contents of the output buffer are sent.

`tp_insspace(n)`
Inserts n blank characters to the left of the current cursor position. The characters on the line to
the right of the cursor position are shifted right by n positions.

```

tp_reset()
```

This function sends a token to the terminal immediately by flushing the output buffer after enqueuing the token. It causes the terminal to return the contents of the shared data structure. This includes the token, the cursor position and the mode parameter array. This allows the host to restore its parameters to the same values as the terminal has stored. This resynchronisation would be used when the host program thought that its parameters had been corrupted or were out of date.

```

.tp_scroll(w, r1, r2, nr)
```

Scrolls the specified region of the current window up or down by nr characters depending on w. If w is given the value 'u' the scrolling is upwards on the screen. If w is set to 'd', the scrolling is downwards. The row numbers R1 and R2 specify the rows between which scrolling will occur. Scrolling is always the full width of the current window.

The operations delete line and insert line can be precisely specified using this scrolling function. This is the only way to perform these operations.

```

tp_session(req)
```

This is the screen management session control function. When req is set to zero, the beginning of a session is specified, and when req is given the value one, the end of a screen management session is specified. Note that nesting of sessions is not allowed! A call to this routine causes the terminal manager to be signalled to go into screen management mode and to return a packet to indicate the success of this. The session function waits for this packet to return and checks the packets contents. If the session setup is successful, the routine opens the terminal data base and accesses the correct entry for the terminal type specified by the terminal manager. If all is well, the routine returns 1, if an error of any sort occurs, zero is returned, and the host should not continue with any terminal communications.

```

tp_setcsr(r, c)
```

The function simply sets the cursor to the coordinates (r, c) where r is the row and c is the column to be positioned at. Remember the numbering of the rows and columns of the screen is from zero the the terminal manager specified maximum in each direction.

```

.tp_setfunc(seqn, str)
```

The setfunc function binds a condition at the terminal manager to an action. The function bound to the condition can be any sequence of the primitives available at the terminal manager.
These primitive functions are described in appendix C. The function is passed a parameter to describe the sequencing order of the predicate, this specifies the order of evaluation of all of the bindings specified. The sequence numbers need not be specified in order, any integers between 0 and 99 can be used as sequence numbers, thus a maximum of 100 conditions can be specified.

The string passed as the second parameter is of the form:

"predicate | consequent"

Thus the string associates a predicate that specifies a condition that must be met before the consequent is performed. There is a special format for this string and a set of available predicates and consequents. It is not possible to specify any boolean condition here, only the standard predicate functions can be used. A full description of the predicate string and the functions available is given in appendix C.

tp_setmode(idx, val)

This sets the mode parameters. The protocol often requires that the terminal manager set the value of a mode parameter. This changes the state of the session, or changes the mode of the terminal manager. This functions sets the parameter idx to the value val.

tp_setup()

Initialises the terminal manager, host application buffers, open diagnostic files, etc.. This function must be called before any screen management session. The function returns true if successful.

tp_shutdown()

Shuts down the terminal manager, measures statistics and reverts the terminal connection to line mode.

tp_string(str, n)

A character string (type = "char *") can be sent to be displayed upon the terminal screen by this function. Given a pointer to a character array (string) and the maximum number of characters to be displayed (n), the characters of the string are transmitted to the terminal manager. If the string contains less than n characters, and ends with a null character, only the characters in the string will be displayed.

tp_tabs(w, n)

Sets and resets tabs in the terminal manager. If w is set to one, a tab is set at column n, if w is given the value zero, the tab at column n is removed whether it already existed or not.
tp_token(req)

Passes the control token to the terminal manager. This is done to give control to the terminal manager to allow it to process the users keystrokes. The control token is appended to the tail of the output buffer and the packet is transmitted to the terminal.

tp_update(rtop, rbot, clef, crit, bsz)

Sets the region specified by the rows rtop and rbot and the columns clef and crit to be the current update window. This defines the rectangular update window limits. The final parameter, bsz, specifies the length of the terminal manager editing buffer. This should be set to the maximum length of the line editing buffer of the host program. All in-line editing is done locally to the terminal manager so it is necessary to know the length of the editing buffer especially when transmitting lines of data.

tp_window(w, n)

If w is set to one, the current active window is set to the saved window n. In this way the host application may receive input from a number of different windows simultaneously displayed upon the screen, by switching between them using this function. If w is set to zero, all the fields are deleted.
Miscellaneous Functions.

tp_eventcode(name)
This function returns the event code corresponding to the event name given. The name is a character string at most 6 characters long. The value returned is obtained from the terminal event data base.

tp_eventname(code)
This function returns the name of the event code given. The code is an integer value, the name returned is character string at most 6 characters long. The name returned is obtained from the terminal event data base.

tp_eventp(code)
This function returns 1 if the event code given is in the terminal event data base for the terminal in question, and 0 if is not in the data base. This function is used to determine an event that can be bound to a terminal manager function that is required. The terminal identifier is know to from the session initialisation when this function was informed of the identification number. This is completely hidden from the applications programmer, who should not be concerned with this detail.

tp_query(idx)
This function allows the state of a mode array element to be queried. The function returns an integer value that is the current value of the mode array element specified. The input buffer is not consulted at all during this operation, it is merely a convenient way of safely finding a stored mode array value. If an illegal index is specified, zero is returned.
Appendix E:

Results from SSMP and DSMP experiments.

A series of experiments were performed to measure the performance of both the DSMP and SSMP protocols. Four tests were performed in order to get a picture of the overall performance of the protocols in different situations. The Chef text editor was used as the host application program for all of these experiments. The experiments were designed to be general tests of various features of the protocols.

To obtain these results, files of commands were transferred from a Macintosh terminal to each version of the Chef editor. The files were made with a special keystroke recorder built into DSMP that wrote all keystrokes to a file. From here, these files were loaded into the Macintosh and then sent back to each of the programs (playback) as if the contents of these files were being entered by the user. This allowed for identical tests to be made for both programs.

Better results may have been obtained for a larger sample that than used for the following experiments because the terminal manager initialisation negotiation would have less significance. This initialisation data can often involve more than 200 characters. The results below are sufficient for the simple comparison that is required. The measurements made below are as follows:

- **Cmd. tokens** - These are the commands that are passed between the host and the terminal. They control the operation of the protocol.
- **Cmd. parameters** - These are parameters to the commands above. They are specially encoded integers that describe the action of a command. Commands have a variable number of parameters
- **Data chars** - The screen image data and other data related to the actual contents of the screen image.
- **Total chars** - All of the characters sent between the host and the terminal. This is the most important measure for comparison.
- **Host wakeup improvement** - This is the factor by which the host interrupts are reduced by the protocols. This is not specific to one of the protocols since it relies on the keystrokes made and the number of packets sent between the host. The number of packets sent is independent of the protocol. It only depends on the events at the terminal.
- **DSMP / SSMP transmission ratio** - This is the ratio of data transmitted between the host and the terminal for the two protocols. It indicates the fraction of SSMP transmitted data for which DSMP is also transmitting.
- **DSMP transmission improvement** - This is the improvement in the amount of data transmitted for a packet switching network (X.25 assumed). It assumes that for the data entry experiments (tests 1 and 2), every user keystroke causes a packet to be sent to the host and another to return. Every packet is assumed to have 9 bytes of control information. The examples given ignore acknowledgements and other packets that may be sent. This measurement can only be made
for the first two experiments because the data that would be sent across the network for an editing job could not be determined easily for the directly connected version.

- **SSMP transmission improvement** - This is the same measure as described for DSMP above.

**Test one:**
This test simply entered a large amount of unformatted data into the editor. No editing functions were used, and no blanks or large sequences of single characters were present in the data. This data was in the format of source code for a text formatting program.

<table>
<thead>
<tr>
<th>keystrokes</th>
<th>packets to host</th>
<th>cmd. tokens</th>
<th>cmd. params</th>
<th>data. chars</th>
<th>total. chars</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>DSMP to host</td>
<td>64</td>
<td>72</td>
<td>1665</td>
</tr>
<tr>
<td></td>
<td></td>
<td>from host</td>
<td>83</td>
<td>199</td>
<td>1665</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SSMP to host</td>
<td>250</td>
<td>394</td>
<td>775</td>
</tr>
<tr>
<td></td>
<td></td>
<td>from host</td>
<td>179</td>
<td>651</td>
<td>791</td>
</tr>
</tbody>
</table>

Host wakeup improvement........... 1698 / 35 = 48.5
DSMP / SSMP transmission ratio... 2352 / 2543 = 0.92
DSMP transmission improvement.... (9 x 2 x 1698) / (9 x 35 + 2352) = 11.5
SSMP transmission improvement.... (9 x 2 x 1698) / (9 x 35 + 2763) = 9.4
**E.3 Experiment Results**

**Test two:**
The data for this test was a program written in the 'C' programming language. A system utility program was chosen at random and used in exactly the same way as for test one. This program had large indentations thus allowing the text compression facilities of DSMP to be tested.

<table>
<thead>
<tr>
<th>keystrokes ..........</th>
<th>2745</th>
</tr>
</thead>
<tbody>
<tr>
<td>packets to host ......</td>
<td>107</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>cmd. tokens</th>
<th>cmd. params</th>
<th>data. chars</th>
<th>total. chars</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSMP to host</td>
<td>203</td>
<td>216</td>
<td>1534</td>
<td>3703</td>
</tr>
<tr>
<td>from host</td>
<td>227</td>
<td>559</td>
<td>2640</td>
<td>5179</td>
</tr>
<tr>
<td>SSMP to host</td>
<td>754</td>
<td>1186</td>
<td>2149</td>
<td>8385</td>
</tr>
<tr>
<td>from host</td>
<td>539</td>
<td>2163</td>
<td>2165</td>
<td>9951</td>
</tr>
</tbody>
</table>

Host wakeup improvement...........  = 25.7
DSMP / SSMP transmission ratio... = 0.44
DSMP transmission improvement.... = 10.6
DSMP transmission improvement.... = 5.3

**Test three:**
This test measures the performance of a genuine editing session. This session includes the entry and editing of a text file. The file has a variety of editing operations performed upon it while it is being entered, and frequent movements around the file are made.

<table>
<thead>
<tr>
<th>keystrokes ..........</th>
<th>1016</th>
</tr>
</thead>
<tbody>
<tr>
<td>packets to host ......</td>
<td>55</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>cmd. tokens</th>
<th>cmd. params</th>
<th>data. chars</th>
<th>total. chars</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSMP to host</td>
<td>79</td>
<td>113</td>
<td>658</td>
<td>1618</td>
</tr>
<tr>
<td>from host</td>
<td>202</td>
<td>334</td>
<td>631</td>
<td>2092</td>
</tr>
<tr>
<td>SSMP to host</td>
<td>453</td>
<td>717</td>
<td>2064</td>
<td>5802</td>
</tr>
<tr>
<td>from host</td>
<td>305</td>
<td>1278</td>
<td>2201</td>
<td>6778</td>
</tr>
</tbody>
</table>

Host wakeup improvement...........  = 18.5
DSMP / SSMP transmission ratio... = 0.28
Test four:
Test four was made to edit an existing 'C' program. This is a program chosen at random that is edited in a typical way. This includes a lot of scrolling and minor changes on lines all the way through the program. It is designed to be similar to a typical programmer fixing syntax errors.

keystrokes............. 938
packets to host....... 65

<table>
<thead>
<tr>
<th></th>
<th>cmd. tokens</th>
<th>cmd. params</th>
<th>data. chars</th>
<th>total. chars</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSMP</td>
<td>94</td>
<td>131</td>
<td>388</td>
<td>1522</td>
</tr>
<tr>
<td>from host</td>
<td>933</td>
<td>1555</td>
<td>4656</td>
<td>12,962</td>
</tr>
<tr>
<td>SSMP</td>
<td>207</td>
<td>415</td>
<td>334</td>
<td>2154</td>
</tr>
<tr>
<td>from host</td>
<td>589</td>
<td>3077</td>
<td>7807</td>
<td>18,490</td>
</tr>
</tbody>
</table>

Host wakeup improvement......... 938 / 65 = 14.4
DSMP / SSMP transmission ratio... 1522 / 2154 = 0.71

Average overall results.
The averaged results for the all of the above tests are:

Ave. host wakeup improvement = 25
Comparison ratio (DSMP / SSMP) = 0.6
Ave. DSMP transmission improvement = 11
Ave. SSMP transmission improvement = 7

Packet size measurements.
The average packet size has been calculated from all of the experiments performed above. This measurement is calculated from the total characters sent and the number of packets transmitted, averaged over all of the previous experiments. All measurements are in characters. The average maximum packet size is included here also for comparison. It can be seen that the packet size limit is constraining the amount of data sent in a packet from the host, but not to the host. It may therefore be an improvement if a larger maximum packet size limit was used from the host.
The trigger value is the level in the output buffer that when exceeded, a packet is sent. The minimum packet size sent was less than 10 characters.

**DSMP**
- Ave. packet size to host = 39
- Max. packet size to host = 89
- Ave. packet size from host = 90
- Max. packet size from host = 209

(max. packet size limit = 220, triggered at 200)

**SSMP**
- Ave. packet size to host = 74
- Max. packet size to host = 151
- Ave. packet size from host = 148
- Max. packet size from host = 235

(max. packet size limit = 256, triggered at 220)
Appendix F:

Examples of the Data Encoding Schemes.

This appendix contains an example of the data sent between the host and the terminal for each of the protocols SSMP and DSMP. This is included for interest to demonstrate the encoding and half duplex operations discussed in the report. It is not expected that any reader should be able to decode the data, but it is useful to see exactly how the components of the systems communicate. This data was taken from the channel between the two components, it is not "made up", it is "real".

Refer to the accompanying computer file for this listing
Appendix G:

The Terminal Manager Program.

This appendix contains the program listing of the terminal manager program. This program is written in the "C" programming language. It is written to be as portable as possible. The file names that makeup the modules are:

- dmain - main program
- dpipes - pipe communication
- dqueue - buffering routines
- dpred - predicate tree
- dptok - host token processing
- dscren - screen functions
- dterm - terminal interface

Refer to the accompanying computer file for this listing
Appendix H:

The Host Application Toolbox.

The program listing for the host application toolbox is shown in this appendix. This file shown here contains all of the toolbox routines described in appendix D. Two support files are necessary to control the communication between the host and the terminal. These are included in appendix G. The files are:

- htools - host toolbox functions
- dpipes - pipe communication (see appendix G)
- dqueue - buffering routines (see appendix G)

Refer to the accompanying computer file for this listing
Appendix I: The Host Application Interface.

This appendix contains the program listing for the terminal interface module of the test host application program "Chef". This module performs all the terminal interface operations for the Chef text editor.

hef7b    - Chef screen mode terminal interface
hef4b    - Chef screen mode processing

Refer to the accompanying computer file for this listing
Appendix J:

SSMP Programs.

The programs for the SSMP system have been put together to save space. The terminal manager and host application programs are still completely separate entities, as can be seen from the listing. These programs are very similar in structure and format to the DSMP programs. The files containing the modules of the terminal manager program are:

- temain - main terminal manager program
- pipes - pipe communication
- queue - buffering routines
- proctok - host token processing
- scrfunc - screen functions
- scrextn - experimental extensions
- vdukey - terminal interface

The file that contain the modules of the host application program are:

- smpget - host toolbox input routines
- smpput - host toolbox output routines
- Chef7b - Chef terminal interface

Refer to the accompanying computer file for this listing.