PE-TI-988

TERMINAL SERVICE ARCHITECTURE

March 11, 1982

DATE: March 11, 1982

TO: R & D Personnel

FROM: Dick Munroe, Evelyn Tate

SUBJECT: Terminal Service Architecture

REFERENCE: PE-TI-847, "Canonical Terminal Requirements"

KEYWORDS: Canonical terminal, Virtual terminal, Terminal Services, PDA

ABSTRACT

This is the first draft of a Terminal Service Architecture for Prime. This architecture is the framework for all future terminal support projects to be undertaken by the Terminal Services Development section of the Communications Software department.

This document contains:

o an overview of the architecture detailing its components and their relationships;

o preliminary specifications for services to be provided, within this framework, to satisfy the requirements in PE-TI-847 for basic class terminal support;

o ideas for the application of this architecture to forms and graphics class terminal support.

We expect that this document will eventually be folded into the Prime Distributed Architecture documentation, as the specification for Core Virtual Terminal Services within PDA.

Table of Contents

Preface

This document is the architectural specification for the Core Virtual Terminal Service of PRIME's Distributed Architecture (PDA). The document is organized into three sections:

- Architectural Specification: Sections 1 to 3. Specify the goals of the Terminal Service, an introduction to the structure of the Terminal Service, and a formal architectural specification of each component of the Terminal Service [not presented in Version 1 of the document].

- Application of the Architecture: Sections 4 to 10. While Terminal Services Development is part of the larger PDA effort, we have to balance architectural vision with product needs. At this time, we have chosen to apply the Terminal Service Architecture to solving the issues dealing with support of "character-at-a-time" terminals and intend the first prototype terminal service to solve these issues.

- Proposal for further applications of the architecture: Section 11. Contains a proposal that defines how we intend to apply this architecture toward supporting other types of terminals.

This draft of the specification is incomplete. Sections which talk about either Window Services or the relationship of the Terminal Service to the rest of PDA are still very preliminary; however, most other areas discussed here will not change much in future drafts. [Text in square brackets generally contains ideas that we hope to expand upon in a future draft.] We have chosen to publish the specification in this state, rather than wait until all pieces have been filled in, so that readers can get a good idea of the direction we're proceeding in, and so that current and planned projects can begin to take the architecture into consideration.

1 Terminal Service Overview

A Terminal Service provides an end-to-end, device and connection independent terminal system.

One "end" of the terminal system is a program; the other "end" is a terminal providing an interface to a human user. The "service" provided is intended to ease the development of programs that use the sophisticated capabilities of terminals.

1.1 Goals of Terminal Services Development

Terminal Services Development (TSD) will provide a Terminal Service that is wholly or partially responsible for the following:

- providing programs with the appearence of a "logical terminal" having a standard set of display and keyboard functions and a set of display and keyboard related services, independent of physical terminal-type.
- shielding programs and terminal users from needing to know details about the configuration of the terminal to program connection.
- establishing and breaking connections between a terminal and one or more processes, or between a process and one or more terminals.
- managing the set of services provided by the Terminal Service on behalf of either a program or a terminal operator under guidelines defined by a system administrator.

1.2 Relationship to PRIME's Distributed Architecture

The Terminal Service is the Core Virtual Terminal Services within PRIME's Distributed Architecture (PDA). The Terminal Service relies on at least these services provided by PDA.

- Job management: to assist in terminal-initiated terminal-to-process binding, and to assist in the task-interrupting behavior required by QUIT.
- Naming service: to assist in terminal-to-process binding.
- Interprocess Communication (IPC): to provide communication between components of the Terminal Service.
- Logical I/O (LIO): to define the form of the interface between programs and the Terminal Service.

- o System Administration: to assist in defining the initial relationships between the terminal and the rest of PDA as well as programs running within a PDA environment.

1.3 Relationship to non-PDA systems

The Terminal Service architecture is being defined as an integral part of the larger PDA architecture. However, it is possible that some or all of the Terminal Service must be provided in a non-PDA system, for example today's PRIMOS. Should this occur, TSD will be responsible for developing any and all PDA-equivalent services for the non-PDA systems.

2 Introduction to the Architecture

```
+--------------------+
¦      Program       ¦
+--------------------+
          ¦
+--------------------+
¦  Library Routines  ¦
+--------------------+
          ¦
+--------------------+
¦      AMLC DIM      ¦ (or remote login,
+--------------------+  or DPTX, or ... )
          ¦
+--------------------+
¦ Physical Terminal  ¦
+--------------------+
          ¦
       Terminal
         User
```

Figure 2.1: Today's Terminal Service

Terminal handling in today's PRIMOS environment is awkward. Programs are responsible for:

- o Connection Independence. Programs are connected to terminals via a number of different program interfaces. "Assignable" and "login" terminals, for instance, must be treated very differently at the program interface.
- o Device Independence. Programs must know how to handle different terminals (for example, the PT45, OWL, and WREN) all of which use different escape sequences to express similar functions.
- o Logical Services. Some general terminal-handling services (for example echoing and erase/kill processing) are handled by PRIMOS through various interfaces. Other useful services (for example, selective echoing and "stop printing when screen is full") must be provided by individual programs in an ad-hoc fashion.

As a consequence, user interfaces to terminal services vary from program to program; terminal services vary from program to program; and capabilities of terminals go mostly unused.

The Terminal Service within PDA will provide a simpler and more powerful program interface which will make it easier for programs to get the most out of the terminal. In order to do this, the Terminal Service provides programs and terminal users with the appearance of a "logical terminal" that provides a consistent and unvarying interface to programs and terminal users.

What is a "logical terminal"? How does the Terminal Service make it appear to exist? How do programs and terminal users control its behavior? The remainder of this section is an informal discussion of the architecture which provides these services in the PDA environment.

2.1 The Logical Terminal

The logical terminal is whatever the program sees as being at the end of its read and write character streams. All terminal functionality visible to the program can be ascribed to this hypothetical logical terminal. In practice, of course, there is no single entity which fills this role; the Terminal Service, the physical terminal, and various PDA services all play a part in presenting this end-image to the program.

The logical terminal has:

o a keyboard capable of generating a certain range of inputs; this range defines a set of keyboard services.

o a display which has a certain shape and size, and a certain set of display behaviors; these behaviors define a set of display services.

o a set of parameters which the program can manipulate to control the logical terminal's behavior.

The program's interfaces to read from, write to, and control the logical terminal are consistent and unvarying.

Physical terminals don't always resemble the logical terminal very closely. This is where the Terminal Service Architecture comes in.

2.1.1 Canonical Terminal Services

The first problem is that physical terminals present a wide range of display and keyboard capabilities. The interfaces to those capabilities are far from standard; everyone is aware that different terminals require different escape sequences to get at the same functions.

Therefore, the first step towards a logical terminal is to provide an interface to physical terminal capabilities that does not depend on the physical terminal.

In our model, the interface consists of a set of standard lexemes which represent standard keyboard and display functions. A lexeme is the smallest self-contained unit of terminal language. Lexemes en route from program to terminal are generally interpreted as display requests; lexemes en route from terminal to program are generally encodings of single keystrokes. The lexeme <A> is therefore interpreted as "display an A" on output

```
            +-------------------+
            |      Program      |
            +-------------------+
                      |
                      |
                      |
                      |
        +-----------------------------+
        | Canonical Terminal Service |
        +-----------------------------+
                      |
                      |
                      |
                      |
            +-------------------+
            | Physical Terminal |
            +-------------------+
                      |
                  Terminal
                    User
```

Figure 2.2: The Canonical Terminal Service

or as "the A key was pressed" on input. Similarly, the lexeme <position home> is a unit which is interpreted as a cursor position request or as an indication that a particular cursor key was used. The actual implementation of lexemes may be as messages, character sequences, or some other structure.

The Canonical Terminal Service's job is to translate between these standard lexemes and the language used by the physical terminal. When the program looks at the physical terminal through the Canonical Terminal Service, it sees what we call the "canonical terminal" -- a terminal with simple, standard behaviors which generates and responds to the standard lexemes.

The Terminal Service will provide three classes of canonical terminals:

o   Basic Canonical Terminal. The familiar "character-at-a-time" terminal. The display services of the Basic CT manipulate (write, highlight, erase) characters on a one or two dimensional display. Later in this document we'll be looking at tha Basic CT in detail.

o   Forms Canonical Terminal. This terminal deals with "fields" (groups of character positions) bound together into a "form". The display services of the Forms CT manipulate fields. The Forms CT is nominally capable of a fair amount of local processing.

o   Graphics Canonical Terminal. A pixel based graphics terminal. We don't have much to say about this topic yet, but we know that the Terminal Service must support such a device.

But there are a lot more services to be provided by the Terminal Service. Why are these services not provided by the Canonical Terminal Services?

2.1.2 Window Services

The answer to the previous questions lies in our desire to provide a class of services related to windows. Windows are display regions through which a terminal user can communicate with one or more programs at once.

```
        +--------------------+
        |       Program      |
        +--------------------+
                  |
                  |
                  |
         +-----------------+
         | Window Service  |
         +-----------------+
                  |
                  |
                  |
+-----------------------------+
|                             |
+-----------------------------+
                  |
                  |
                  |
        +--------------------+
        | Physical Terminal  |
        +--------------------+
                  |
              Terminal
                User
```

Figure 2.3: The Window Service

In today's environment, there is only one "logical terminal" associated with each physical terminal; a terminal user can talk to only one process from his terminal.

We would like to do away with this restriction and make it possible to support multiple processes from a single terminal. One terminal user may want to work on several things simultaneously, for example simultaneously running an editor, a compiler, a debugger, while a status program checks for incoming mail messages.

We believe that the best human interface in this environment is one which allocates a separate piece of the display, or "window", to each conversation, so that they don't interfere with each other. The user can watch several things happening at once, direct his input to the process he wants to talk to, make interesting windows bigger, and do lots of other neat things.

Most programs are and will continue to be developed with the assumption that the program "owns" the entire terminal; the compiler won't be aware of the existence of the mail program, or any other programs which happen to be sharing the terminal at the moment. So, the Window Service provides programs with the appearance of a private canonical terminal, one (or more) per program.

This "private canonical terminal" still looks like a canonical terminal to the program; the program can send lexemes to the display, and read lexemes from the keyboard. When the program says "erase the whole display", that indeed happens, as far as the program is concerned. However, the Window Service intervenes to ensure that only the appropriate window gets erased, and not in fact the whole canonical display. The Canonical Terminal Service continues to do its job, unaware that window manipulation is happening above it.

With the Window Service in place, there is no need to require the dimensions of the private canonical display to exactly match the canonical display dimensions. We can let the program "see" a very large or a very small display. The Window Service can take care of mapping that very large display onto the canonical display, through a window.

From the terminal user side, the Window Service provides the mechanism that partitions the canonical terminal into windows through which the terminal user sees these "private canonical terminals".

In the requirements definition for the Terminal Service, the issue of windows was not addressed. Consequently, we're not ready to specify the exact nature of windows in any detail. We are using this as a placeholder until we understand, or get someone else to specify, the requirements for windows and operations upon windows.

Okay, so much for windows, but back to the original question. What happened to all the rest of the services that were supposed to be provided by the Terminal Service? Why aren't those services provided by the canonical terminal?

2.1.3 Logical Terminal Services

```
      +-------------------+
      |      Program      |
      +-------------------+
                |
                |
                |
  +-----------------------------+
  | Logical Terminal Service    |
  +-----------------------------+
                |
                |
                |
       +------------------+
       |                  |
       +------------------+
                |
                |
                |
 +------------------------------+
 |                              |
 +------------------------------+
                |
                |
                |
      +-------------------+
      | Physical Terminal |
      +-------------------+
                |
            Terminal
              User
```

Figure 2.4: The Logical Terminal Service

Once windows are part of the Terminal Service architecture we have a problem deciding where the logical terminal services (erase/kill, line wrapping, etc.) should be provided.

As an example, look at a terminal user who is running FUTIL in one window and EMACS in another. These program want different behaviors, different "personalities" for their logical terminals; for instance, FUTIL wants its logical terminal to handle all echoing, and EMACS wants to disable that feature since it's doing its own version of echoing.

If the Canonical Terminal Service is responsible for echoing, does it echo a particular keystroke or not? Clearly it depends on which logical terminal the current window belongs to; but we've said that the Canonical Terminal Service doesn't know about windows. Suppose the Window Service cleverly changes the Canonical Terminal Service's echoing parameter whenever the user moves the cursor to a different window? That's a little better, but suppose the Canonical Terminal Service is called upon to echo a "clear the entire display" request. It still can't do the right thing without an intimate knowledge of the window setup.

A much simpler solution is to put the "personality" portion of the logical terminal above the Window Service. The Logical Terminal Service is responsible for providing those behavior-modifying services. Like the Canonical Terminal Service, the Logical Terminal Service is unaware of the existence of windows. It does its job thinking that it owns a private canonical terminal; the Window Service makes sure that the window boundaries are observed.

Each logical terminal has its own set of logical terminal services, provided by its own copy of a Logical Terminal Server. Examples of logical terminal services are:

o Echoing -- associating a visual display operation with each keystroke.

o Erase/Kill -- allowing the terminal user to modify his typed input before the program gets it.

o Translation -- allowing the program to converse in EBCDIC, even though the canonical terminal is defined to understand ASCII.

In short, a logical terminal service is anything that alters the program's perception of the behavior of the canonical terminal. A program which was happy with the definition of the canonical terminal, and which wanted to send and receive canonical terminal lexemes with nothing added, wouldn't need a Logical Terminal Service at all.

There will probably be completely different "packages" of logical terminal services, aimed at providing different logical terminal personalities. Later in this document we'll be looking closely at one such package, which is intended to support (with considerable extensions) PRIMOS's "standard" login terminal personality.

Since each program has its own logical terminal, and we've said that different logical terminals, using the same physical terminal, can have different behaviors (echo and no echo, for instance), we have to provide a way for programs to control the logical terminal's behavior.

2.1.4 Parameter Management Service

The Parameter Management Service allows a programs to find out or change the state of its logical terminal. The program hands "read parameter" and "write parameter" requests to the Parameter Management Service, which performs the appropriate service.

```
+-----------------------------------------------------+
|                       Program                       |
+-----------------------------------------------------+
                |                         |
                |                         |
                |                         |
                |                +---------------------+
                |                | Service to manage   |
                |                | parameters          |
                |                | that control ...    |
 +----------------------------+  |---------------------|
 |                            |-| Logical Terminal    |
 +----------------------------+  | Services            |
                |                |---------------------|
                |                          |
                |                          |
       +-----------------+       |---------------------|
       |                 |-----| Window Services     |
       +-----------------+       |---------------------|
                |                          |
                |                          |
                |                          |
+-----------------------------+  |---------------------|
|                             |-| Canonical Terminal  |
+-----------------------------+  | Services            |
                |               +---------------------+
                |
                |
      +-------------------+
      |                   |
      +-------------------+
                |
             Terminal
               User
```

Figure 2.5: The Program managing the Logical Terminal

The program doesn't know about the components inside the Terminal Service; it's interested only in the net effect -- the logical terminal. Within the Terminal Service, however, a particular parameter may be relevant to any one, or more than one, of the three servers. The size of the logical display, for instance, is known to both the Logical Terminal Service and the Window Service; anything having to do with the physical display (for instance, whether it's capable of reverse video) is known only to the Canonical Terminal Service.

The Parameter Management Service is responsible for providing a single program interface for managing the logical terminal. The program never sees the complexity within the Terminal Service.

2.1.5 Summary of the Logical Terminal

Programs and terminal users see the Terminal Service from different perspectives.

From outside the Terminal Service a program sees a "logical terminal". As far as a program is concerned, the "logical terminal" has a certain size, shape, and a characteristic set of behaviors that programs find useful.

On the other hand, the terminal user sees "logical terminals" through windows displayed on his physical terminal. "Windows" (or viewports) allow the terminal user to partition the display portion of the canonical terminal in a way that allows the terminal user to talk to many programs at the same time. "Windows" aside, the "logical terminal" seen by the terminal user is the same object that the program sees, with the same nice features: a certain size, shape, and characteristic behaviors.

These four components of the Terminal Service contribute to the appearance of a "logical terminal".

- Logical Terminal Service: provides the characteristic set of behaviors that programs and terminal users expect. This is the "personality" portion of the logical terminal. We talk about one set of behaviors in detail later.

- Window Service: provides the size and shape of the logical terminal, and manages the windows through which the terminal user views one or more logical terminals.

- Canonical Terminal Service: provides a standard "vocabulary" (lexemes) for manipulating terminals. This is the "physical interface" portion of the logical terminal. The rest of the Terminal Service, and as a consequence all programs outside the Terminal Service, need never know the details of managing particular types of terminals.

- Parameter Management Service: provides the management interface that allows programs to control parameters to modify the behavior of the logical terminal.

So far we've explained how the program controls the logical terminal. But what about the terminal user? We know he'll want to modify the logical terminal's behavior too, to change erase/kill characters, perhaps to control window sizes, and so forth. He can talk to programs through the Terminal Service, but how does he talk to the Terminal Service?

2.2 Human Interface Service

The Human Interface Service is a special program which the terminal user uses to manipulate the parameters of any of his logical terminals.

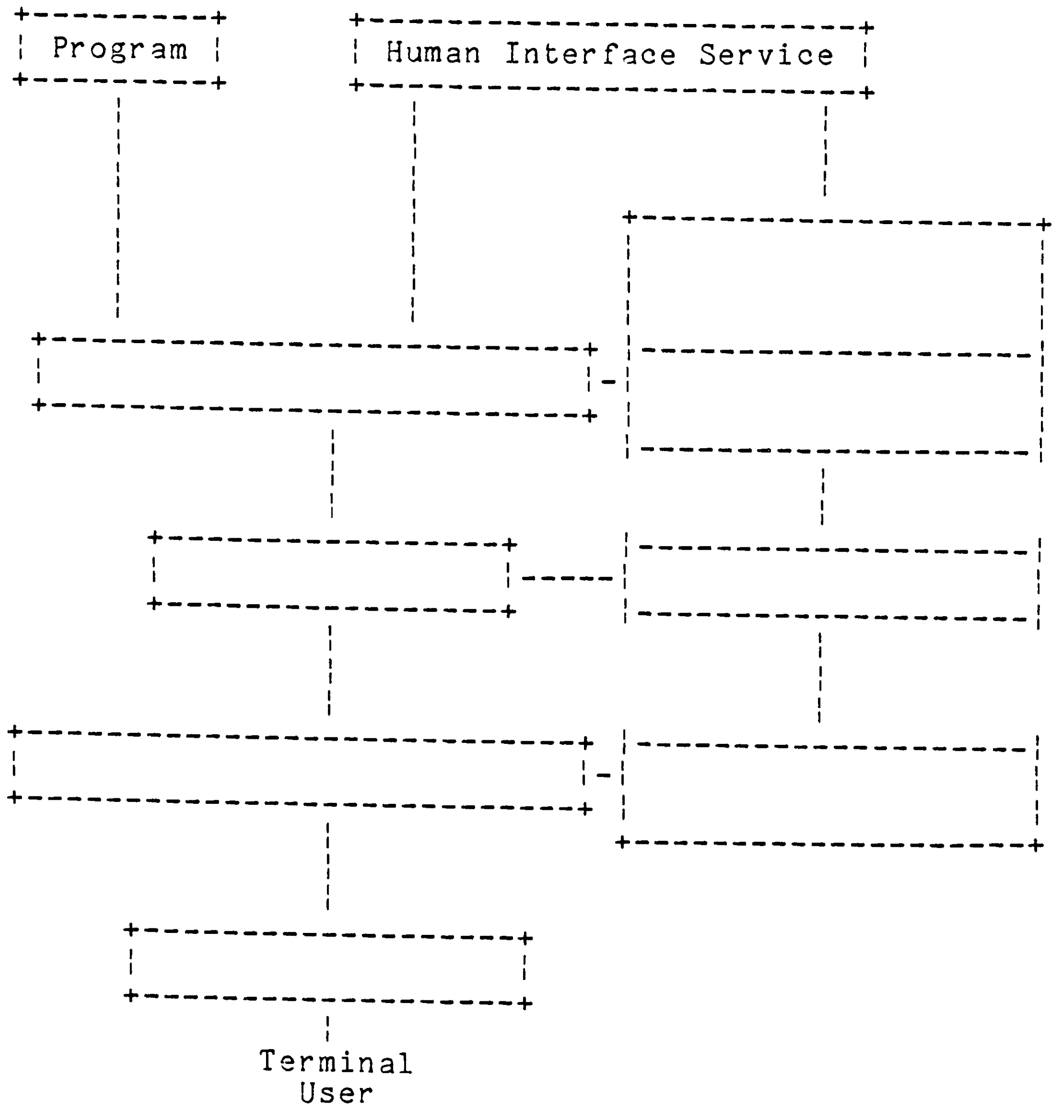

Figure 2.6: The Terminal User managing the Logical Terminal

The Window Service provides a way for a terminal user to converse with many programs through a single canonical terminal. Taking advantage of this facility, we have modeled the Human Interface piece of the terminal service as "just another program" using the services provided by the Terminal Service to carry on a conversation with the terminal user.

The Human Interface Service talks to the terminal user through a logical terminal, just as any other program would. This logical terminal will be windowed onto the canonical terminal, just like any other logical terminal. The Human Interface Service uses the capabilities of the logical terminal (for example, echoing, erase/kill, clear display) to provide an appropriate user interface. The conversation between the terminal user and the Human Interface Service consists of whatever is considered "normal" or "comfortable" for the user: menus, forms, TERM commands, or any other user interface that is considered appropriate. The Human Interface Service invokes the Parameter Management Service to change parameters for <u>any</u> of the user's logical terminals, including the one in use by the Human Interface Service.

But there's still a piece missing. The Parameter Management Service controls <u>a</u> logical terminal on the behalf of the program which uses it. We've just said that the Human Interface Service can control some other program's logical terminal. But we previously said that programs sharing the same canonical terminal don't know about each other and can't see each other's logical terminals. How does the Human Interface Service do it?

<u>2.3 Terminal Service Overseer</u>

To answer that question, we introduce the last component of the Terminal Service. The Terminal Service Overseer provides all services that involve the relationships between streams owned by a single terminal user. There are streams within the Terminal Service, (maybe) streams to Job Manager(s), and the normal streams (e.g. standard in and standard ou) to processes which are clients of the Terminal Service. The Terminal Service Overseer is a "master manager" that manages the connections among components of the Terminal Service and provides the terminal user's "window" into the rest of PDA.

This is the least understood piece of the Terminal Service, because it has an intricate relationship to the PDA environment, which isn't completely defined yet. We can mention some of the services we know the Terminal Service Overseer will provide.

- Terminal Service Initialization: Finding all the correct servers and hooking them together so that the Terminal Service can operate.

- Terminal Session Initialization: Getting a terminal user connected to the right Job Management Service so that he can access the complete PDA environment.

- Terminal Process Stream Management: Getting everything required for a new "logical terminal" set up when a process opens a new stream to the Terminal Service (including the right Logical Terminal Service).

```
       The Terminal Service           .  Prime's Distributed
                                      .  Environment
       +-+  +-----------+             .
       | |  |           |             .
       +-+  +-----------+             . +-- Name Space Services
        |     |   |   |               . |
        |     |   |   |               . |  System Administration
        |     |   |   |               . |  Services
        |     |   |   |               . |
        |     |  +-+ +-------------+  . |  Job Management Services
        |     |  | | | Terminal    |  . |
        |     |  | |-| Service     |-----+ Logical I/O Services
        |     |  | | | Overseer    |  .
       +-------+ |-| +-------------+  .    Inter-Process
       |       |-| |                  .    Communication
       +-------+ | |                  .    Services
           |     |-|                  .
           |      |                   .    Security Services
           |      |                   .
       +-------+ |-|                  .    etc.
       |       |-| |                  .
       +-------+ |-|                  .
           |      |                   .
           |      |                   .
           |      |                   .
       +-------+ |-|                  .
       |       |-| |                  .
       +-------+ | |                  .
           |     +-+                  .
           |                          .
           |                          .
       +-------+                      .
       |       |                      .
       +-------+                      .
           |                          .
```

Figure 2.7: Managing the Terminal Service

- Stream Interrelationships: Providing special clients like the Human Interface Service with the ability to control other logical terminals.

- Security: Keeping unauthorized programs from accessing other logical terminals.

- And probably others we haven't thought of yet.

2.4 A short tour of the Terminal Service

"Nothing is what it appears to be! All is illusion and chaos is loose on the world..."

from Waldo & Magic, Inc. by R. Heinlein.

If you have made it this far, there is lots more stuff that follows. Here are a few brief scenarios describing how we think all this will work within PDA. Unfortunately, even a brief scenario is a couple of pages long. If you don't intend to read any further, these scenarios will leave you with a good idea of the "flavor" of the Terminal Service. If you do intend to read further, these examples should make the rest of the document easier to understand.

2.4.1 Getting started

We assume that, at the very beginning, the PDA System Initialization Service has started up a Terminal Service Overseer, and that that, in turn, has started up the appropriate Canonical Terminal Server (basic, forms, etc.) for each terminal, as specified in some system profile. This is the minimal state for an inactive terminal. (Notice that all terminal I/O is made canonical from the very beginning -- the Canonical Terminal Service is always there to keep terminal-type-specific data out of the system.)

This scenario shows what happens to get the whole Terminal Service assembled when a terminal becomes active.

- o The terminal user sits down at terminal XYZ and presses a key on the physical keyboard.
- o The Canonical Terminal Service receives the terminal's input for that keystroke, converts the input to canonical form (a lexeme), and emits the lexeme upward ...
- o Where the Terminal Service Overseer gets the lexeme. The Overseer realizes that terminal XYZ is trying to enter the system. The Overseer first connects the Canonical Terminal Service to a Window Service.
- o Next, the Overseer starts up the appropriate (according to some profile) Human Interface Service and tells it to connect itself to terminal XYZ. The Human Interface Service selects a set of logical terminal services and invokes Logical I/O to connect a named stream to XYZ. The Overseer fields the CONNECT request at the terminal end, creates a logical terminal, and plugs in the specified Logical Terminal Server.

- The Overseer goes through the same steps to get a PDA Job Management Server hooked up to the same terminal, with its own (possibly different) Logical Terminal Server.

- The terminal user has entered the system.

All this has happened in response to that first keystroke. The terminal user has at his command a Job Manager, waiting for him to log in, and a Human Interface Server, waiting for him to set or examine terminal service parameters (including the erase/kill characters to be used during the log in dialog).

2.4.2 A new logical terminal

Let's look in a little more detail at a step we glossed over in the previous scenario -- creating a new logical terminal. This will be a frequent operation for the Terminal Service; it might happen whenever the Job Manager starts up a new process, to run from the same terminal; when the system mail service, having a message for a user, opens a stream to the terminal that user's logged in at; or when EMACS requests a new window to display another file buffer.

- A process connects a named stream (or a pair of unidirectional named streams?) to the object XYZ.

- The PDA Logical I/O service interprets that request and invokes the PDA Session Establishment Service, which finds XYZ in the global name space and informs the Terminal Service Overseer for XYZ about a new incoming stream.

- The Overseer allocates a new logical terminal data base for XYZ, plugs the default Logical Terminal Service into the new stream between itself and the process, starts up a Parameter Management Service to go with it, and passes its end of the stream to the Window Service. The Overseer also sets up a default window allocation for this logical terminal. (We have no idea how the default window should look.)

- The Window Service repaints the Canonical Display to make room for the new window, which is initially empty.

- The new logical terminal is ready for use.

The process may want to manipulate its logical terminal parameters (change the echo behavior, or some such) before it starts sending data to the display.

2.4.3 Data out and in

The final example will follow some lexemes through the Terminal Service. This won't by any means be a complete description of the work done at each level; it's just intended to give the flavor of what might happen where, and why. For this example, we'll assume the Basic Canonical Terminal and the "general purpose" Logical Terminal Service have been selected.

First, look at what happens to the graphic character "A" between program and terminal.

- o Within LIO: The LIO mechanism takes a semantic request for "A" from the program, and maps that into the lexeme <A> to be passed to the terminal service. The format of the program's semantic request hasn't been determined at this writing; most likely it will be just the ASCII character "A" in a buffer referred to by a LIO WRITE statement.

- o The lexeme first encounters the Logical Terminal Server (LTS) at "level 3" of the Terminal Service. The server applies some set of tests and transformations to output lexemes. In this example, it might test to see whether "discard output" has been requested (in which case the <A> would be thrown away), or whether any lexeme transformation is required (such as happens for control character expansion). If the <A> passes all tests, it will be passed onward.

- o At level 2 (the Window Server (WS)): This server checks to see whether this process's output is currently windowed onto the canonical display (via the Basic Canonical Terminal Server) and whether this display request can be satisfied within this window. If the active position is in a different window, the <A> may need to be preceded by a <position> lexeme to get it to appear in the right region of the canonical display.

- o At level 1, the Basic Canonical Terminal Server (CTS): This server, on receipt of the <A>, does whatever is necessary to (a) get the character A to appear at the current active position of the display, and (b) advance the active position in accordance with standard rules (including what is to happen at the boundaries). For most terminals, sending an ASCII "A" out on an asynchronous line will do the trick; however, there may be extra translation (ASCII to EBCDIC, or lower to upper case) or special protocol handling (for an SDLC terminal) at this level, to handle idiosyncrasies of particular terminals.

Now, watch the same "A" as it progresses from terminal to program.

- At level 1 (CTS): The terminal-dependent keystroke for "A" is mapped into the the lexeme <A>. Usually there's nothing to this, but remember there could be translation or special protocols involved here.

- At level 2 (WS): This server determines which of (possibly) several process input streams should get this lexeme.

- At level 3 (LTS): This server, on receipt of the <A> lexeme, applies some set of input tests and transformations. One sample service includes echoing (should this lexeme be echoed? if so, as what?). An echo will cause the appropriate lexeme(s) to be shipped to the display (via the level 2 server, as usual). If the program has invoked the "character-at-a-time" data forwarding service, the <A> will be sent off to the program; if not, the Logical Terminal server may hold onto it until some other condition (such as end of line) is met.

- Within LIO: The LIO mechanism will convert the <A> lexeme into some semantic representation understood by the program. (Probably just an ASCII "A" character in a READ buffer.)

For a slightly more interesting case, let's follow a "move cursor up" request from program to terminal.

- Within LIO: The Logical I/O mechanism takes the semantic request (that the cursor be moved up one position) from the program, and maps that into the lexeme <position up> to be passed to the terminal service. We don't know yet the format of the terminal's request; it could be a sequence of ASCII characters, or a special character in a Prime-defined character set, or an LIO POSITION function which looks similar to a WRITE function, or a device-specific "terminal position" call.

- At level 3 (LTS): This server performs the same set of checks on the <position up> lexeme as it did for the <A>.

- At level 2 (WS): This server performs the same set of checks as it did for the <A>. If the <position up> would move the cursor outside of the allowed window, the server would probably replace the <position up> lexeme with some different flavor of <position>.

- At level 1 (CTS): When this server gets a <position up> lexeme, it does whatever is necessary to cause the cursor to move up one position on the physical display (subject to the standard boundary violation rules). For some terminals, the server can just invoke the terminal's "position up" operation -- (e.g. for a PT45 the server would send the right two-character escape sequence). For some terminals, the server may have to invoke an explicit "position to <x,y>" operation, providing a row and column number.

Now, watch the same "move cursor up" as it progresses from terminal to program.

- At level 1 (CTS): When the terminal user hits the "position up" key (usually labeled with an arrow) on, say, his PT45, the terminal generates an escape sequence which is identified by the Basic Canonical Terminal server. The server collects the characters and maps them into the single lexeme <position up>.

- At level 2 (WS): Routes the lexeme to the right process input stream.

- At level 3 (LTS): This server applies the same tests and transformations to the <position up> lexeme as it did for the <A>. This includes echoing (should a <position up> be echoed?), data forwarding (should a <position up> cause forwarding?) and so forth. This server could even be told to discard positioning lexemes completely, if the program preferred not to handle them.

- Within LIO: The LIO mechanism will convert the <position up> lexeme into some semantic representation understood by the program. We don't know what the format would be yet, but it might be a character sequence, a special character, an on-unit invocation, or whatever.

Seems awfully easy, doesn't it. Come back in December, 1982 for a demo of the prototype.

3 Architectural Specification

[This section will be the formal definition of the architectural responsibilities of each component of the Teminal Service Architecture. We won't fill this section in for Draft 1. Most, if not all, of the material comprising the content of this section is present in the sections below. Extracting that material, editing the resulting changes, incorporating the draft 1 review comments, and expanding the application sections appropriately will be the primary content of draft 2.]

4 Concepts

This section defines and discuss concepts and terms used throughout the document. [It isn't complete yet.]

4.1 Modeling strategy

This document is not an internal design document for the Terminal Service. This document does defines the framework that partitions the design and implementation. Specific physical implementation requirements will influence how the framework is actually implemented.

4.1.1 Servers

For the purpose of this model, we have partitioned the Terminal Service into a number of individual servers. This strategy serves two purposes. First, it allows us to partition the overall Terminal Service into smaller and simpler sets of services (a terminal-independence service, a parameter-controlling service, and so forth), which can be individually discussed and understood more easily than the whole. Second, it provides us with a reasonable set of guidelines for distributing the overall Terminal Service in a PDA environment. The separately-defined servers are good candidates for distribution.

4.1.2 Workers

Workers are asynchronous algorithms within a server. Each worker is big enough to be interesting and small enough to be described clearly. Workers are simply a convenience for partitioning and describing the problems that must be addressed within a server. Workers are not intended to be a detailed design specification of how to solve the problems. We think of workers as mini-servers that cooperate within the context of a bigger server. Other ways do exist to approach the modelling problems for which we use the workers.

4.1.3 Levels

We use the term "level" occasionally throughout this document to refer to the components of the Terminal Service which act on and transfer data -- "terminal traffic" -- between program and the physical terminal. These levels are numbered in order of their proximity to the physical terminal; the Canonical Terminal Service resides at level 1, the Window Service at level 2, and the Logical Termial Service at level 3.

4.2 Lexemes

Lexemes are the smallest self-contained units of terminal language. Programs will communicate with the Terminal Service through a protocol which consists of an exchange of lexemes. The existence of lexemes may, however, be hidden from the application programmer by appropriate higher level interfaces. Servers within the Terminal Service also use lexemes to communicate among themselves.

Lexemes are language units, not "commands". All components of the Terminal Service must be prepared to receive and parse all lexemes, but their meaning and interpretation may vary throughout the Terminal Service.

4.2.1 Lexeme syntax

General syntax for lexemes is

```
<lexeme-name [{parameter [, ...]}]>.
```

Examples of well-formed lexemes are:

```
<oregano>
<fruit {banana}>
<make salad {lettuce, carrot, tomato}>
```

This syntax is used to differentiate lexemes from the "characters" that they represent, i.e. the character "A" and the lexeme <A> are different objects.

We have not chosen a format for internal representation of lexemes, but it must have these characteristics.

- o Regular. Any server should be able to recognize and parse all lexemes, even unknown ones.
- o Capable of inclusion of arguments whose data type is dependent on the lexeme name. Column numbers and visual attributes are examples of arguments we want to support within lexemes.
- o Extensible. We would like to support an arbitrarily large lexeme vocabulary.

We know of several existing implementations of lexeme-like objects in other systems; these include:

- o European Virtual Terminal Protocol "messages": lexeme = length + code + code-specific arguments.

- ANSI/ISO terminal control sequences: lexeme = introducer + code + code-specific arguments + terminator.

4.2.2 Standard lexemes

We expect a wide variety of lexemes to be defined for different versions of the Terminal Service. For instance, the vocabulary for a forms class terminal will include lexemes for field definition concepts which are not relevant to basic class terminals. However, there is a set of standard lexemes which is of general usefulness and which should be always supported. These include lexemes for all ASCII characters, and a few miscellaneous ones.

4.2.2.1 Graphic Lexemes

There is a lexeme for each ASCII graphic character, 95 in all. They include letters (e.g. <A>, <a>), numbers (e.g. <1>), punctuation and other special characters (e.g. <!>, <^>), and blank (<blank>).

The notation <graphic> will be used to represent any graphic lexeme.

4.2.2.2 Format Effector Lexemes

There is a lexeme corresponding to each of the six ASCII format effectors. We have added two, <new line> and <space>, which are implicitly defined in ASCII as alternate interpretations of other characters. (See the section "About ASCII and Terminals" for more explanation.) The eight format effector lexemes are:

- <BS> or <back space>
- <HT> or <horizontal tab>
- <LF> or <line feed>
- <VT> or <vertical tab>
- <FF> or <form feed>
- <CR> or <carriage return>
- <NL> or <new line>
- <SP> or <space>

The notation <format effector> will be used to represent any format effector lexeme.

4.2.2.3 The Bell Lexeme

The <bell> lexeme corresponds to the ASCII control character "BEL".

4.2.2.4 Control Lexemes

These are lexemes which correspond to all the rest of the characters in ASCII which have no specific application to terminal functions. They are generally used for communications control, device control, and information separation. There are 26 control characters; they are the first 32 characters of ASCII, octal :0 through :37 (minus the format effectors and BEL, which are terminal related functions) and the DEL character (octal :177). We will refer to them using either their ASCII names (e.g. <NUL>, <ETX>, <DC1>) or the more familiar control-letter terminology (e.g. <control-@>, <control-C>, <control-Q>.

The notation <control> will be used to represent any control lexeme.

4.2.2.5 Special Purpose Lexemes

o <repeat {number}>

This lexeme will cause the previously-received lexeme to be repeated {number} more times.

o <transparent {byte-string}>

The byte-string is passed through without any modification by the interpreter of this lexeme. Bytes are 8-bit quantities.

o <nil>

A lexeme with no effects. The <nil> lexeme can be discarded at any time with no unexpected side effects.

4.3 About ASCII and terminals

This section provides a little background relevant to our use of ASCII, the "American National Standard Code for Information Interchange", ANSI standard X3.4-1977.

4.3.1 Why ASCII

Since ASCII is the commonly accepted character standard, we have adopted it (with modifications described below) as the standard lexeme vocabulary for terminals. We are not ruling out support for other character sets; in fact we fully expect to support many character sets in addition to ASCII. However, we cannot yet justify replacing ASCII as the "standard" character set with any of the "extended ASCIIs" currently proposed or in use outside Prime. This may change if, for example, Prime makes a commitment to Office Automation large enough to justify adopting the proposed ISO document-processing character standard ("Teletex") as our official internal character set, replacing ASCII.

4.3.2 Line feed and new line

The ASCII standard contains the following statement about the "line feed" format effector.

> "Where appropriate, this character may have the meaning 'New Line' (NL), a format effector that advances the active position to the first character position on the next line. Use of the NL convention requires agreement between sender and recipient of data."

Prime has adopted the NL convention for internal use. We do this by always interpreting the LF character with NL semantics. This makes the plain LF function unavailable.

We want to make both functions available to programs and terminal users. Rather than creating one lexeme which has two different interpretations depending on context, we have created two separate and unambiguous lexemes, <LF> and <NL>. Most Terminal Service users will use <NL>, but <LF> will be available as well.

The ASCII standard also allows the other vertical format effectors, vertical tab and form feed, to be interpreted as moving the active position to the first column of the new line. Since neither character is used much at Prime now, we've made the arbitrary decision which seems most useful, which is to always include the "first column" semantics when the <VT> and <FF> lexemes are interpreted.

4.3.3 Space and blank

The ASCII standard contains the following statement about the "space" graphic character.

> "SP (Space). A graphic character that is usually represented by a blank site in a series of graphics. The space character, though not a control character, has a function equivalent to that of a format effector that

causes the active position to move one position forward without producing the printing or display of any graphic."

These two meanings -- produce a blank character position, and move the active position forward one position -- are equivalent in the usual case of printing text on a previously-blank display. But when the positions affected are not already empty, the meanings are different: the first replaces any existing character with a blank, while the second is a non-destructive positioning operation. An example of this confusion can be seen in the way RUNOFF underlines text on a terminal; it uses spaces to move the cursor to the text to be underlined, but a CRT interprets the request as blanks, erasing all preceding text in the process.

To avoid this ambiguity, we have created two separate lexemes: <blank> is a graphic character, and <space> is a format effector (the reverse of <back space>). Most Terminal Service users will use <blank>, but <space> will be available as well.

5 The Canonical Terminal Server (Level 1)

Level 1 is the "bottom" level of the Terminal Service. This level is the closest to the physical terminal. The server at this level is called a Canonical Terminal Server, or CTS; it handles the mapping of standard display and keyboard operators onto physical terminal's displays and keyboards.

A client of the CTS invokes standard display services by sending standard lexemes to the CTS. The display effects of the standard lexemes are defined in terms of their effects on an abstract display, called the canonical display (CD). The CTS provides standard keyboard services by generating standard lexemes which represent keyboard input. The standard lexemes will appear to have been produced by an abstract keyboard, called the canonical keyboard (CK). The combination of canonical display and canonical keyboard is called a canonical terminal (CT). The Canonical Terminal Server handles any operations necessary to provide its clients with the appearance of a canonical terminal. The Canonical Terminal Server also emulates the the canonical terminal on physical terminals (devices consisting of a paired display and keyboard).

The CD and the CK will be defined to behave in ways that are representative of most, but not all, existing displays and keyboards. Some work must usually be done, by any implementation of a Canonical Terminal Server, to emulate the CD and CK using existing equipment. The amount of space allocated for data structures, and work done to translate lexemes into physical terminal language, will be directly affected by how closely the physical display and keyboard match the CD and CK. If

- o the display understood lexemes and provided all of the display effects defined for the CD, and
- o the keyboard generated the standard lexemes in response to keystrokes, and
- o the terminal had management interfaces to set and read the parameters defined in this section

then the physical terminal would be an implementation of a Canonical Terminal Server, and we would not have to write separate software to support the CT. If PRIME were to build a "FINCH" terminal that met the above requirements, we could achieve a substantial savings in work and space, with a corresponding increase in overall system performance.

Because of the wide variation in capabilities of physical terminals, we have defined three classes of canonical terminal.

- o Basic canonical terminal. This class is modeled upon the familiar character-oriented terminal which has either a scrolling or a page display.

- Forms canonical terminal. This class of terminal has a display which is organized into fields (groups of contiguous character positions) which together constitute a form.

- Graphics canonical terminal. This class of terminal has a pixel display.

The architecture allows the existence of a separate CTS for each class of canonical terminal. The canonical display and canonical keyboard for each class will be defined to reflect the capabilities of actual terminals of that class. Standard lexemes will be defined to represent display and keyboard functions of each class.

This section of the Terminal Service Architecture document describes in detail the Basic CT and the Basic Canonical Terminal Server (BCTS). The other terminal classes will be discussed briefly in the later section "Proposals for further applications of this architecture".

5.1 Invoking the BCTS at level 1

[How this happens ... when you would use the basic CT package rather than some other (e.g. forms) CT package. This function is intimately associated with the Overseer Initialization Service. The "appropriate" set of CT services is probably set up once, for all time, by the Overseer Service. We talk about how the appearance of various CT services might be achieved in the section "Proposal for further applications of the architecture".]

5.2 About scroll and page mode

BCTS provides two distinctly different flavors of service, called scroll mode and page mode. These have different display behaviors because the underlying model of a display is different in each mode. To explain why we did this, here's some of the background.

5.2.1 Real world analogues of each flavor

A scroll class display functions like a teletype. The characters are displayed at the cursor (or active) position. The cursor advances one character position after each character. When a scroll class display sees a "new line" character or a "carriage return/line feed" character sequence, the display (usually paper) "scrolls" up by one line so that the cursor appears at the beginning of a fresh and empty line. Previous lines can no longer be written on (the paper doesn't scroll down) although the operator may still be able to see them. A "glass teletype", which is a CRT on which new lines are entered at the bottom and old lines scroll off the top of the screen, also has scroll class functionality.

A page class display functions like a sheet of paper with <m> lines each of <n> characters. The cursor (or active) position can be moved to any of the <m>*<n> positions on the page. When characters are displayed on a page class display, the cursor behaves in much the same way as for a scroll class display. However, the "new line" character or "carriage return/line feed" character sequences do not guarantee an empty line in a page class display. Once characters are displayed on a page they must be explicitly "erased".

5.2.2 Why the models are conceptually different

At first, a scroll class display appears to be a degenerate case of a page class display, where the number of lines <m> happens to be one. It might therefore appear that there is really only one kind of display. This is almost but not quite the case.

The key to the difference between scroll and page class displays is in the scrolling action of the scroll class display. A "new line" character, or its equivalent, a "carriage return/line feed" character sequence, causes the scroll mode terminal to physically "scroll" -- the platen advances and the "print head" is positioned to the beginning of a new line (e.g. of paper on a hardcopy terminal). The new line is guaranteed to be empty, "fresh", regardless of the program's past behavior. Only the new line of paper is accessible for display functions -- the platen cannot roll backwards and the print head cannot move backwards. Therefore any characters printed before the "new line" character are unchangeable. (This is the critical point -- if the platen and print head can move backwards, it's a page display.) The length of the "paper" is considered to be unbounded and each "new line" character character has the same effect as any of the previous ones.

On a page display, which is explicitly two dimensional and bounded, a "new line" character is a positioning operator which moves the active position or cursor to the beginning of the next of the <m> available lines. There is no concept of "scrolling"; nothing changes except the active position. The active position can be moved "up", so previously-written positions on previous lines can be revisited. The line the cursor is on following a "new line" character contains whatever the program previously put there (which may be empty if that line had been previously erased by an "erase line" or "erase display"). Since the page display has a finite number <m> of lines, "new line" character requests will work until there are no "new" lines. A "new line" character request that cannot work, e.g., the cursor at the bottom of the page, is treated in the same way as a boundary-violating positioning request.

Obviously, the canonical displays exhibit two different display behaviors. Some physical displays will be able to emulate both the scroll and page CD, some will be able to emulate only the scroll or page CD. The decision to emulate either the scroll or page CD (or both!) on a particular physical display is an implementation issue.

Since the scroll and page canonical displays are different in intent, switching from one type of canonical display to another results in the creation of new canonical displays, previous information about the CD is lost.

5.2.3 Why we need both models

The scroll class display is the least common denominator of display functionality. Scroll class features are commonly used by our software and are currently supported by PRIMOS for all displays. (Almost all real displays can operate as scroll class displays and we emulate scroll class for those which don't, for example IBM 3270 terminals.)

Terminals with displays having only scroll class capabilities, e.g., the ASR / KSR model 33, are no longer the norm. Almost all terminals have displays with two-dimensional cursor positioning, the prerequisite for a page class display. However, as the least common denominator, a scroll class display is assumed by most of today's PRIMOS programs -- including compilers, command processors, some editors, and most tools.

A page class display can present a better human interface than a scroll class display (for example in screen editors). However, a standard program interface to page class displays is not currently available. As a consequence not many of today's PRIMOS programs take advantage of page class displays.

Therefore, both classes of display must be available: a scroll class display for compatibility with the past, and a page class display for exploitation of the present.

5.3 The Basic CT Services

The primary source for the definition of Basic Canonical Terminal services has been the set of requirements presented in the Canonical Terminal Requirements document. We have added a few things (e.g. the separate cursor and active position) which we feel are useful at this level.

This section provides only an overview of the services which the Basic Canonical Terminal Server will provide. Detailed specifications of all services, including the interactions among them and the management interfaces used to invoke and control them, will be developed for a separate document, the Terminal Service

Functional Specification.

BCTS clients (either programs or other components of the Terminal Service) will access these services through lexemes. BCTS will perform a display service (e.g. display graphic, erase display) when it receives the appropriate lexeme from a client; and it performs a keyboard service (e.g., application function key) by generating a lexeme to be provided to the client [awkward]. The lexemes associated with services are listed along with the service descriptions.

5.3.1 Appearance of a standard display

BCTS behaves as though it were performing display requests on a standard display, called the canonical display (CD). Any differences in behavior between the canonical display and the actual physical display in use will be hidden by BCTS; as far as the client is concerned, the physical display IS a CD.

The canonical display is an array of character positions, each of which can hold a displayable character and can have an associated highlight. (See "Character Display Services" and "Highlight Services".)

Associated with the canonical display is an active position (AP) which is the CD location at which the next display operation will take place. Most canonical display operations refer to or change the active position (or both). The active position can be controlled by the program. (See "Position Services".)

The canonical display comes in two flavors: scroll and page. The scroll CD is a one dimensional array -- a "line" with {m} columns. The page CD is a two dimensional array -- a "page" with {n} lines, each of {m} columns. BCTS always supports a scroll CD, and may support a page CD. The program can determine whether a page CD is available, and can select page or scroll mode, through the Parameter Management Service.

The size of the CD (number of columns for a scroll CD, number of columns and lines for a page CD), is determined by BCTS. The program can access these values through the Parameter Management Service.

5.3.2 Appearance of a standard keyboard

BCTS behaves as though it were connected to a standard keyboard which had a standard set of keys; this standard keyboard is called the canonical keyboard (CK). Any differences in keyboard repertoire between the canonical keyboard and the actual physical keyboard in use will be hidden by BCTS; as far as the client is concerned, the physical keyboard IS a CK.

The canonical keyboard has about 160 keys, which are grouped into categories according to the type of lexeme the keys generate. (Most real keyboards have far fewer keys; they use shift and control keys to make it possible to generate several lexemes with the same key, appropriately modified.

o Graphic keys generate graphic lexemes. Graphic lexemes represent all 95 ASCII graphic characters, including "blank".

o Format effector keys generate seven of the eight format effector lexemes. (There is no key which generates the <space> lexeme.)

o The Bell key generates the <bell> lexeme.

o Control keys generate the control lexemes that represent those ASCII control characters which are neither format effectors nor BEL. (The fact that most real keyboards have a single "control" key which modifies the actions of graphic keys so that they emit <control> lexemes is irrelevant for the definition of the canonical keyboard.)

o Display function keys generate lexemes that represent the guaranteed canonical display functions.

o Application function keys generate application function lexemes. These lexemes have no predefined association with any CT concepts or display effects. Application function lexemes may have meaning for applications, but do not for the CT.

o Terminal-specific function keys are keys on physical keyboards that have no other meaning to the CK; they generate the <terminal-specific> lexemes. Terminal-specific keys are like application function keys in that they have no predefined association with any CT concepts or display effects. Terminal-specific keys are different from application function keys in that their presence is dependent on the physical terminal type and is not guaranteed by the architecture.

Each canonical key generates a standard lexeme. A list of canonical keys and their corresponding lexemes is is figure 5.1.

Graphic, format effector, bell, control, and display function keys are guaranteed to exist on all canonical keyboards. Application function and terminal-specific function keys may be available; the program can inquire about their availability through the Parameter Management Service.

Key Class	Key Name	Lexeme Produced
Graphic	"a", ")", etc.	<a>, <)>, etc.
Format Effector	Back Space	<BS>
	Horizontal Tab	<HT>
	Vertical Tab	<VT>
	Form Feed	<FF>
	New Line	<NL>
	Carriage Return	<CR>
	Line Feed	<LF>
Bell	Bell	<BEL>
Control	"NUL", "DC1", etc.	<NUL>, <DC1>, etc.
Display Function	Position Up	<position up>
	Position Down	<position down>
	Position Left	<position left>
	Position Right	<position right>
	Position Home	<position home>
	Erase Character	<erase character>
	Erase to Beginning of Line	<erase to beginning of line>
	Erase to End of Line	<erase to end of line>
	Erase Line	<erase line>
	Erase to Beginning of Display	<erase to beginning of display>
	Erase to End of Display	<erase to end of display>
	Erase Display	<erase display>
	Insert Character	<insert character>
	Insert Line	<insert line>
	Delete Character	<delete character>
	Delete Line	<delete line>
Application Function	Application Function (0...16)	<application function {0...16}>
Terminal-specific function	Terminal dependent	<terminal-specific function {0...m}>

Figure 5.1: The Canonical Keyboard by key class

5.3.3 A standard communications protocol

BCTS communicates with clients through standard PDA streams which carry standard BCTS lexemes. BCTS provides any translation (e.g. ASCII to EBCDIC), repackaging (e.g. SDLC framing), and protocol (e.g. "request to send" -- "clear to send") required for conversation with the physical terminal. The client uses only standard PDA stream interfaces.

5.3.4 Character display service

The character display service allows a program to display any ASCII graphic character at the current active position. (The active position is modified either implicitly or explicitly by the Position service.)

There are 95 graphic characters (including blank); each is represented by a graphic lexeme, for example <a>, <A>, <=>, <blank>. The notation <graphic> will be used to represent any graphic lexeme.

The character display service is invoked by each of the <graphic> lexemes.

BCTS currently supports only ASCII graphics. In the future, this service may be extended to support additional graphics, including line drawing characters, word processing characters, APL characters, and VIDEOTEX characters.

5.3.5 ASCII Format Effector services

The format effector service allows a program to move the active position on the display in a way that conforms to the ASCII definitions for format effectors.

BCTS provides these services in response to <format effector> lexemes.

- <BS> or <back space>
  The active position is moved backward one position on the same line.

- <HT> or <horizontal tab>
  The active position is advanced to the next predetermined character position ("horizontal tab stop") on the same line. Horizontal tab stops are defined every 8 positions on a canonical display line.

- <LF> or <line feed>
  The active position is advanced to the same character position on the next line.

- <VT> or <vertical tab>
  The active position is advanced to the first character position on the next predetermined line ("vertical tab stop"). Vertical tab stops are defined every 8 lines.

- <FF> or <form feed>
  The active position is advanced to the first character position of the first line of a new "form". BCTS determines what constitutes a form for this terminal type.

- <CR> or <carriage return>
  The active position is moved to the first character position of the same line.

- <NL> or <new line>
  The active position is moved to the first character position of the next line.

- <SP> or <space>
  The active position is moved forward one character position on the same line.

The effect on the active position of hitting a display boundary (bottom, left, or right) is in most cases the same as that described under the "Position" service. The exceptions are for the <NL>, <LF>, <VT> and <FF> format effectors when the canonical display is a scroll CD. In this case, these format effectors by definition cause the "scrolling" action associated with scroll terminals; the effect on the (single line) scroll CD is to re-initialize all its character positions to default (blank, unhighlighted) values.

The format effectors are most useful with a scroll CD, since the scroll CD is a model of the kind of terminal for which format effectors were invented. When the canonical display is a page CD, the actions associated with format effectors are similar to or duplicated by actions associated with other, more general position services. BCTS continues to supports the format effectors for page mode because we believe that all ASCII terminal constructs should be supported in basic class.

5.3.6 Alarm service

The alarm service lets the program invoke the canonical terminal's "bell" to get the terminal user's attention. It's invoked by the <bell> lexeme.

5.3.7 Position services

The position service allows the program to move the active position to another point on the canonical display. It has three flavors.

- Implicit active position advance.
  (invoked by all <graphic> lexemes)
  Whenever a character is placed on the display, the active position is automatically moved forward by one. (See "Character Display Service".) The active position is also moved as a side effect of the erase line / erase display services. (See "Erase Services".) This kind of positioning is supported on both scroll and page CDs.

- Format effector positioning.
  (invoked by all <format effector) lexemes)
  This kind of positioning is supported on both scroll and page CDs, although the actions of vertical format effectors (NL, FF, etc.) are different on the different types. (See "ASCII Format Effector services".)

- Explicit active position movement.
  (invoked by all <position> lexemes)
  BCTS provides a small number of primitive positioning operations which cause either absolute or relative active position movement. This kind of positioning is available only when the canonical display is a page CD.

  - <position horizontal {column}>
    The AP is moved to the specified column on the current line.

  - <position vertical {line}>
    The AP is moved to the same column on the specified line.

  - <position absolute {line, column}>
    The AP is moved to the specified position on the display.

  - <position home>
    The AP is moved to the first position on the first line of the display.

  - <position up>
    The AP is moved up one position.

  - <position down>
    The AP is moved down one position.

  - <position left>
    The AP is moved one position to the left.

- <position right>
  The AP is moved one position to the right.

The Canonical Display has boundaries. If the positioning service is requested to move the active position outside the boundaries of the CD,

- the position request is ignored (the AP stays where it was),

- the component (column or line) of the AP which has been violated becomes undefined, and

- any successive lexemes received by BCTS whose operation would reference the undefined component of the current active position will be ignored until the active position is positioned back within the boundaries of the CD by a <format effector> or <position> request.

For example, a character display request is received when the AP is at the last column of a scroll display. The character is displayed, but the implicit advance is ignored, and all successive display requests will be ignored until a <CR>, <NL> or <FF> (which don't need to know the current column number to be performed). As another example, a <position up> request is received when the AP is on the top line of a page display. The <position> request is ignored, and all successive display requests which refer to "current line" will be ignored -- <position down> and <HT>, for instance. But a <position vertical> or <position home> will be accepted and display can continue normally again.

The program can read the current value of the active position through the Parameter Management Service.

5.3.8 Erase services

The erase service allows a program to erase part or all of the display. A character position which has been erased contains a blank instead of whatever character it held before, and has no highlighting.

BCTS supports these primitive erase services, invoked by the specified <erase> lexemes.

- <erase character>
  The character position at the active position is erased.

- <erase to beginning of line>
  All character positions preceding the active position on the current line are erased.

- <erase to end of line>
  All character positions from the active position (inclusive) to the end of the current line are erased.

- <erase line>
  All character positions on the current line are erased. The active position is moved to the first position on the current line.

- <erase to beginning of display>
  All character positions preceding the active position on the display are erased.

- <erase to end of display>
  All character positions from the active position (inclusive) to the end of the display are erased.

- <erase display>
  All character positions on the display are erased. The active position is moved to the first position of the display.

BCTS provides these erasure services (invoked by the <erase> lexemes) only when the canonical display is a page CD. On a scroll CD, positions can be erased with a combination of <format effector>, <define highlight {none}>, and <blank> lexemes.

5.3.9 Highlight services

The highlight service allows a program to specify the highlighting to be applied to character positions on the display. Particular highlight settings are called "visual attributes" or "attributes".

- Highlighting: bold, blink, underline, reverse video, etc.

- Color: foreground and background.

- Font or typeface

- Probably others to be defined.

The program invokes the service by sending a <define highlight> lexeme. BCTS will then apply this highlighting to each character position into which it writes a displayable character, until the next <define highlight> lexeme. For example, the lexeme sequence

```
<define highlight {blink}>
<A> <B> <C>
<define highlight {bold, underscore}>
<D> <E> <F>
```

will result in characters A, B, and C being displayed so that they blink, and characters D, E, and F displayed in boldface and with underscores (but not blinking).

The lexeme <define highlight {none}> will cause subsequently displayed characters to have no highlighting at all.

The program can read the set of highlights currently in effect through the Parameter Management Service.

5.3.10 Insert/delete services

The insert/delete services allow programs to open or close space on the canonical display. The program will use <graphic> lexemes to put graphic characters into the space at the active position opened by the insert service. Graphic characters will be removed at the active position by the delete service.

- o <insert character>
  All characters from the active position (inclusive) to the end of the current line are moved forward one position. The character position at the active position is erased. The character previously at the end of the line is lost.

- o <insert line>
  All characters on all lines from the current line (inclusive) to the end of the display are moved down one line. The current line is erased. The previous contents of the last line are lost.

- o <delete character>
  All characters following the active position on the current line are moved backward one character. The last character position of the line is erased. The character previously at the active position is lost.

- o <delete line>
  All lines following the current line on the display are moved up one line. The last line of the display is erased. The previous contents of the current line are lost.

BCTS supports insert/delete services only when the canonical display is a page CD.

5.3.11 Visible cursor

Normally, a visible cursor is associated with the active position. The cursor thus serves as visual feed back for the terminal operator. Under some circumstances programs may wish to update the canonical without disturbing the terminal operator by also moving the cursor (most notable is supporting "windows", discussed in the next section). The cursor service allows the program to update the canonical display without moving the cursor.

o <bind active position>
The active position is bound to the cursor. As a result the cursor position becomes the active position and when the active position moves, the cursor also moves.

o <unbind active position>
The active position is unbound from the cursor. Moving the active position does not move the cursor.

BCTS supports the "separate cursor" service for both scroll and page CDs.

The program can read the current cursor position through the Parameter Management Service. The program will also be able to modify the visual representation of the cursor through the Parameter Management Service.

5.3.12 Miscellaneous services

[Transparent, repeat, terminal-specific function]

5.4 BCTS Interfaces

This section provides an overview of interfaces used by BCTS clients to access BCTS services.

5.4.1 Display/Keyboard interfaces (lexemes)

BCTS generates and responds to individual lexemes. It reads a lexeme at a time from its input stream and performs the associated display service. It generates a lexeme at a time, which it write to its output stream, in performance of a keyboard service.

Figure 5.2 lists the lexeme vocabulary used by BCTS. An "X" in the column labeled "Scroll CD" or "Page CD" indicates that that lexeme invokes a display service for that kind of canonical display; no "X" means that that lexeme is ignored. An "X" in the column labeled "CK" means that the canonical keyboard can generate that lexeme; no "X" means that the CK will never

Scroll CD	Page CD	CK	Lexeme name
X	X	X	all <graphic> lexemes
X	X	X	<backspace>
X	X	X	<horizontal tab>
X	X	X	<line feed>
X	X	X	<vertical tab>
X	X	X	<form feed>
X	X	X	<carriage return>
X	X	X	<new line>
X	X		<space>
X	X	X	<bell>
		X	all <control> lexemes
	X	X	<position horizontal {column}>
	X	X	<position vertical {line}>
	X	X	<position absolute {line, column}>
	X	X	<position home>
	X	X	<position up>
	X	X	<position down>
	X	X	<position left>
	X	X	<position right>
	X	X	<erase character>
	X	X	<erase to beginning of line>
	X	X	<erase to end of line>
	X	X	<erase line>
	X	X	<erase to beginning of display>
	X	X	<erase to end of display>
	X	X	<erase display>
	X	X	<insert character>
	X	X	<insert line>
	X	X	<delete character>
	X	X	<delete line>
		X	<application function {selector}>
X	X		<define highlights {selector}>
	X		<bind active position>
	X		<unbind active position>
X	X	X	<repeat {number}>
X	X		<transparent {byte string}>
		X	<terminal-spec. function {selector}>
			<nil>

Figure 5.2: Lexemes used by BCTS

produce that lexeme.

5.4.2 Management interfaces (parameters)

[List of all parameters and values]

5.5 About the BCTS server

A functional design for BCTS will include descriptions of

- data structures: objects which maintain state, describe services, or otherwise hold information needed by BCTS to perform services;
- workers: little servers which perform the BCTS services in response to the receipt of display lexemes or during the generation of keyboard lexemes.

[We're not sure that this type of functional design material belongs in this architecture document as it's currently scoped, but here it is anyway.]

5.5.1 Structures used in BCTS

In this section we describe the data structures used by BCTS.

5.5.1.1 The Display Structure

The Display Structure is a one-dimensional (scroll mode) or two-dimensional (page mode) array of cells. BCTS uses it to keep track of all changes to the physical display, because BCTS may need to refer to the current state of the physical display in order to perform some requested operations (e.g., "insert character" may require copying existing characters on the display to new locations). A particular implementation of BCTS may not need to maintain a display structure if the physical terminal's behavior is close enough to that of the canonical display.

5.5.1.1.1 The "Cell"

The addressable unit of the line and page is the cell. A cell has several fields. Operations that manipulate cells may cause changes in any or all fields of the cell.

- Character
  The character field of each cell contains an encoded representation of a single graphic character. When the cell is mapped onto a physical display, the appropriate

graphic will be made visible in the display position associated with this cell. The contents of this field will be interpreted as an ASCII character value.

- Attributes
  The attributes field of each cell contains the visual characteristics to be applied to the character when it is made visible on a display. Attributes may include highlighting information ("blinking", "reverse video"), color information (possibly foreground and background colors), and font/typeface information ("gothic", "italic").

- State
  The state field of a cell contains flags which describe cell as a whole. One such state variable is "undefined", which means that the character and attribute fields may not be interpreted.

A cell which contains a blank character and a default ("no highlighting") set of attributes is said to be "erased".

When a cell is bound to a position on the physical display, the cell's graphic character is displayed, with the cell's attributes, at the bound position. Within reason, the Canonical Display Worker will emulate graphics and attributes that are not directly supported by the physical terminal.

5.5.1.1.2 The "Line"

A one-dimensional array of cells is a line. A line, when operated on by the proper rules, is the scroll display structure. The line is necessary in the page display structure to define the semantics of some lexemes, e.g., insert line or delete line.

Lines have these characteristics.

- They have a fixed number, <n>, of cells or length.
- The cells within a line are numbered by column numbers from 1 to <n>.
- A line is considered to be horizontal, as we are used to thinking of display lines on a terminal, so that left refers to lower numbered columns, and right refers to higher numbered columns.
- Two cells of a line are considered adjacent if their column numbers differ by one.

The length of a line usually reflects constraints introduced by some physical display.

5.5.1.1.3 The "Page"

A two-dimensional array of cells is called a page. A page is the page display structure, the scroll display structure never uses a page.

Pages have these characteristics.

- A page has a fixed number of lines, <m>, or height. A page whose height is 1 is indistinguishable from a line.
- Cells within a page are numbered by a coordinate pair of line and column number: column numbers are defined above, line numbers range from 1 to <m>, where <m> is the number of lines in the page.
- A page is considered to be a vertical arrangement of lines, as in a display of a CRT. In this context, above refers to lower line numbers, and below refers to higher line numbers.
- Two cells of a page are considered adjacent if their line number is the same and their column numbers differ by one.

The size of a page usually reflects constraints introduced by some physical display.

5.5.1.2 The Keyboard Structure

This is a table that defines the relationships between physical keystrokes and lexemes.

5.5.1.3 The active position

The active position is the location of a cell in the data structure. It is BCTS's internal representation of the active position associated with the canonical display.

The active position is expressed as the coordinates of a cell. For a scroll display, which is explicitly one-dimensional, the active position is a single column number {x}. For a page display, which is two-dimensional, the active position is a line and column pair {y,x}.

5.5.1.4 The cursor

When the display structure is mapped onto a physical display, the cursor provides the terminal user a visual representation of "where things will happen next". The cursor is maintained within BCTS as a structure consisting of:

o a cell position (the format is the same as for the active position), and

o a description of the visual representation of the cursor (a cursor glyph). [We don't know what a good set of cursor glyphs should be.]

5.5.1.5 The attribute register

The attribute register is a data structure which contains a set of visual attributes. As <graphic> lexemes are bound to to cells in the display structure, the contents of the attribute register are copied into the attribute field of the cell.

5.5.1.6 The cursor glyph register

The cursor glyph register is a data structure which contains instructions for the display of the cursor. Whenever the cursor is moved to a new cell, the contents of the cursor glyph register are applied to that cell. [This will probably be a highlight value such as reverse video or underscore; however, we're not sure we want to rule out use of graphics as cursor glyphs.]

5.5.1.7 The previous lexeme register

The previous lexeme register contains the lexeme received before the current lexeme. BCTS will use this register when processing a <repeat> lexeme. The initial value of the previous lexeme register is <nil>.

5.5.1.8 The scroll/page mode toggle

5.5.1.9 The transparency toggle

5.5.1.10 Boundary violation toggles

5.5.1.11 Hardware echo toggle

5.5.1.12 Cursor-AP binding toggle

5.5.1.13 Overstrike/replace toggle

5.5.2 Workers

The model of the Basic Canonical Terminal Server consists of two workers ("workers" are defined in the Concepts section of this document). These workers are:

```
          |     Lexemes To / From        ^
          |      Higher Levels of        |
          |    the Terminal Service      |
          v                              |
+----------------------------------------------+
|         v                              ^     |
|  +------------+              +------------+  |
|  | Canonical  |              | Canonical  |  |
|  | Display    |              | Keyboard   |  |
|  | Worker     |              | Worker     |  |
|  +------------+              +------------+  |
|         v                              ^     |
+----------------------------------------------+
          |                              ^
          |     Commands To / From       |
          |     the Physical Device      |
          v                              |
```

Figure 5.3: Inside the Canonical Terminal Server

- o The Canonical Display worker. This worker receives lexemes from servers at higher levels of the architecture, modifies the internal structures (display structure, active position, etc.) as indicated by those lexemes, and emulates the CD on physical displays.

- o The Canonical Keyboard worker. This worker takes input from the physical keyboard, converts the input into lexemes, and passes the lexemes to servers at higher levels of the architecture.

The Canonical Display and Canonical Keyboard workers of the Basic Canonical Terminal Server are discussed here in separate sections.

5.5.2.1 The Canonical Display Worker

The Canonical Display Worker applies these rules as it receives lexemes from above.

- o Upon receipt of a <transparent {byte-string}> lexeme, the worker interprets the {byte-string} as one or more eight-bit bytes to be sent directly to the physical terminal. The bytes will not be transformed or interpreted, but they may be packaged according to the requirements of the physical terminal protocol. The display structure, active position, cursor, and other structures maintained by the Canonical Display Worker are unchanged during the <transparent> operation. A program using <transparent> lexemes is responsible for resolving any resulting difference between the physical and canonical displays.

- o Any lexeme which is not defined for this type (scroll or page) of canonical display, will be ignored. (See Figure 5.2 for a list.)

- o If the HORIZONTAL-FAULT flag is SET, a lexeme whose associated operation requires any reference to the current column value of the active position will be ignored. A similar rule concerning line position applies if the VERTICAL-FAULT flag is set.

- o If the lexeme passes the above tests, then the service associated with that lexeme is performed, and the display structure, active position, (possibly) cursor position, and other structures are changed as appropriate.

- o If the performance of the lexeme assigned a new column value to the active position, the HORIZONTAL-FAULT flag is reset. A similar rule applies to VERTICAL-FAULT.

- o Whether the lexeme was performed or ignored, a copy is stored in the previous lexeme register before the next lexeme is examined.

[The lexeme-by-lexeme detailed descriptions of display rules, which were present in the first draft, have been removed to a different document. We believe that those descriptions can now be derived in a straightforward manner from the BCTS service definitions.]

5.5.2.2 The Canonical Keyboard Worker

The Basic Canonical Terminal Server provides a Canonical Keyboard (CK) to go along with the CD. The Canonical Keyboard Worker makes it possible for many different physical keyboards to emulate the CK.

The management of the canonical keyboard is very simple. The rules are simple because canonical keystrokes have no display side effects and the Canonical Keyboard Worker maintains no state which could modify the "meanings" of keys -- the sequence and timing of keystrokes have no effect whatsoever on the rules by which lexemes are generated.

The rules applied by the Canonical Keyboard Worker are these.

- o If the TRANSPARENCY flag is SET, the Canonical Keyboard Worker forms incoming keyboard data into eight-bit bytes, which are then placed one at a time inside <transparent {byte-string}> lexemes.

- o Otherwise, the Canonical Keyboard Worker recognizes an incoming canonical keystroke and constructs the corresponding lexeme.

- o The Canonical Keyboard Worker sends each lexeme, as soon as it's generated, to servers at the upper levels of the Terminal Service. It does not buffer lexemes.

In all cases, there is a simple one-to-one correspondence between canonical keys and lexemes: the "A" graphic key generates the <A> lexeme, the Position Home key generates the <position home> lexeme, and so forth. See Figure 5.1, "The Canonical Keyboard", for the key-to-lexeme correspondence.

5.5.3 Notes to the Implementor

(not part of the architectural specification)

This section provides guidelines and suggestions to BCTS implementors about how canonical terminal emulation might be accomplished on physical terminals.

5.5.3.1 Display notes

The implementor of a Canonical Display Worker to support a particular physical terminal will decide which classes (scroll, page, or both) of canonical display to support, and how to map the canonical display onto the terminal's physical display. In general, implementors will follow these guidelines.

- o For any terminal at least scroll CD capabilities must be implemented.
- o The Canonical Display Worker must accept and understand all of the standard lexemes. Those lexemes which are defined to have no effects on the CD (e.g. <application function>) may be discarded.
- o The implementor should attempt to provide all CD capabilities for any terminal. However, some CD capabilities may not be reasonably or satisfactorily emulated for a particular physical display. (Examples of features that may not be reasonable to emulate on some terminals are the "Insert Character" operation and the "blinking" attribute -- imagine these features emulated on an ASR-33.) If this situation occurs, those features will not be available to a program using the canonical display; that information <u>will</u> be available to programs. The evaluation of "reasonable" and "satisfactory" will be left to the judgment of the implementor.
- o Because of windowing considerations it's almost never useful to provide a scroll CD if the PD is capable of page... but it's mandatory anyway for those cases (special for the future but ubiquitous now) where one CD = one window = one process.

5.5.3.2 Keyboard notes

The mapping of physical keystrokes to canonical keystrokes is an implementation issue for each physical terminal and can not be formally specified here. In general, however, implementors will follow these guidelines.

- o All physical keys should map onto canonical keys.

- o Physical keys that are labeled the same as canonical keys, e.g. "TAB" or "PF1", should have the obvious correspondence to canonical keys. Otherwise, the implementor should choose reasonable mappings. For example, the key labeled "clear field" on a physical terminal might be chosen to correspond to the "erase to end of line" canonical key.

- o Physical keys which are not otherwise useful should map onto the set of "undefined" canonical keys. For example, keys labeled "AUX ON" or "PRINT" might become "undefined" keys.

- o Physical keys whose actions are invisible to the CK worker cannot and must not be used as canonical keys. For example, a "HOME" physical key whose action is strictly "local" (it moves the cursor but the CK Worker is not notified that the keystroke occurred) cannot be used as a CK key.

- o The terminal operator must be able to generate <u>all</u> CK keystrokes, excluding the class of undefined keystrokes. If necessary, a standard keystroke sequence will be defined by which the terminal user can mimic any missing canonical key.

- o The CK is assumed to have keys that generate <NL>, <LF>, and <CR> lexemes. Most physical keyboards have only a RETURN and a LINE FEED key, and it is customary for the RETURN key to invoke both "carriage return" and "new line" semantics (though not at the same time). If necessary, a standard protocol will be defined by which the terminal user can mimic all three canonical keys using only two physical keys.

- o If the physical terminal echoes all keystrokes itself, and this feature cannot be managed or disabled, the HARDWARE ECHO flag should be SET. The entity which would normally handle echoing (either part of the Terminal Service, or a client program itself) will be able to find out about this condition and change its behavior accordingly. Such terminals won't be very satisfactory as canonical terminals; we can't help much.

6 The Window Service (Level 2)

[This section should be viewed as a placeholder for some ideas about windows rather than as a formal statement. We have not studied this area much yet.]

The Window Service manages the association of many logical terminals, in use by one process or by several different processes, and a single canonical terminal. The terminal user can thus watch and control several services at once from his terminal.

The Window Service provides a "switchboard" that allows:

o many logical displays to be associated with a single canonical display, and

o lexemes from the canonical keyboard to be associated with one of many logical keyboards.

In the special case of "one process to one terminal" (and "one logical terminal to one canonical terminal") connection, the switchboard function of the Window Service is trivial -- the routing is automatic. In the general case, wherein several independent processes share access to one physical terminal, the server must

o route each incoming lexeme from the canonical keyboard to the correct process input stream, and

o handle each display request from a process's output stream subject to any constraints imposed by the sharing of the display.

In the most general case, each process output stream will map onto a region of the display called a window.

We (deliberately) haven't spent much time thinking about the Window Service. Window Services are not required in order to satisfy any of the basic canonical terminal requirements (see PE-TI-847), so we have left this piece for last. However, we are so sure that a Window Service will be a big part of PRIME's terminal service offering in the future that we have made sure to reserve its place in the architecture.

This section of the document will outline a set of services and a set of interfaces to those services which we currently believe should be included in a Window Server. However, the reader should view this as a first pass at a set of requirements for such a service, and not as a formal statement of services to be provided.

```
+-----------+   +-----------+           +-----------+
| Process 1 |   | Process 2 |           | Process 3 |
+-----------+   +-----------+           +-----------+
     LIO             LIO              LIO          LIO
 (stream A)      (stream B)      (stream C)   (stream D)
      |               |               |            |
      |               |               |            |
+----------+    +----------+    +----------+  +----------+
| Logical  |    | Logical  |    | Logical  |  | Logical  |
| Services |    | Services |    | Services |  | Services |
+----------+    +----------+    +----------+  +----------+
      |               |              |             |
      |               |              |             |
+-----------------------------------------------------------+
|     \               | Window Service  |            /      |
|      \              |                 |           /       |
|       \----------+  +-----+      +-----+  +-------/        |
|                  |        |      |        |                |
|            Display/Keyboard Switchboard                     |
+-----------------------------------------------------------+
                              |
                              |
                  +-------------------+
                  |       Basic       |
                  | Canonical Terminal|
                  |      Service      |
                  +-------------------+
                              |
                              |
               +-------------------------+
               |    Physical Terminal    |
               |                         |
               |  +--------+---------+   |
               |  | Window | Window  |   |
               |  |   C    |    B    |   |
               |  +--------+-+-------+   |  <-- Display
               |  | Window   | Window|   |
               |  |   A      |   D   |   |
               |  +----------+-------+   |
               |    +-----------+        |
               |    | ::::::::: |        |  <-- Keyboard
               |    +-----------+        |
               +-------------------------+
                           |
                           |
                   terminal operator
```

Figure 6.1: Many Processes; 1 Physical Display

6.1 Services

6.1.1 Many-to-one mapping

Each program to Terminal Service stream corresponds to one logical terminal. The Window Service allows one or many logical terminals to be associated with a single canonical terminal. These logical terminals may belong to a single process, or to some number of processes.

6.1.2 Windows for logical displays

Each logical terminal has a logical display. A logical display is seen on the canonical display only through a window maintained by the Window Service. A window is a two dimensional region of the canonical display, of arbitrary size and shape. Each program display request is examined by the Window Service; it will cause changes to the canonical display only if the part of the logical display affected by the display request is currently windowed onto the canonical display.

The dimensions of the logical display need have no relationship to the dimensions either of the canonical terminal's display or of the window through which the logical display is viewed. The size of the logical display will be chosen for the convenience of the program; the mapping of the logical display to the canonical display, through a window, is for the convenience of the terminal user.

At any time, some (perhaps all) of the logical displays will be windowed onto the canonical display, while some (perhaps none) will not be seen at all. Logical displays can exist without windows.

6.1.3 Active logical keyboard

Each logical terminal has a logical keyboard. Through the Window Service, the canonical keyboard is mapped onto a single logical keyboard at any time. The terminal user absolutely controls this mapping. At any time he can request that his input be switched from the currently active logical keyboard to the logical keyboard of any other logical terminal associated with the canonical terminal.

6.1.4 Operations on windows

The terminal user will be able to:

- create a window to be associated with a logical terminal, making that logical display visible on the canonical display;
- delete a window, making the logical display formerly seen through that window now disappear from the canonical display;
- move a window to a different region of the canonical display;
- expand a window, making a larger piece of the logical display visible through it;
- shrink a window, making a smaller piece of the logical display visible through it.

The terminal user requests these services at will through the Human Interface Service.

6.1.5 Window "scrolling"

If the logical display is larger than the window currently allocated to it, the terminal user will be able to view only a portion of the logical display at any time. The terminal user can change this visible portion by a "scrolling" operation, which appears to move the logical display "behind" the window so that different sections come into view. The logical display can be so moved up, down, left, or right.

6.1.6 Window identification

For every window on the canonical display, the terminal user will be able to readily identify the logical terminal associated with it. (This might be done by having a constant label on each window, or by having the Terminal Service display a window's identification when explicitly requested to do so.)

6.1.7 Selecting a logical keyboard

The terminal user can specify which logical keyboard is to receive his keyboard input, by pointing to a window on the display, or by supplying the name of a logical terminal. The latter method is required if the desired logical terminal isn't currently windowed onto the canonical display.

6.1.8 Overlaying windows

A new window can be created, or an old one moved, so that it partially or completely obscures another window. The obscured window will again become completely visible when the overlapping window is moved, or if the obscured window is itself moved.

6.1.9 Synchronous/asynchronous update

The terminal user can specify either synchronous or asynchronous display behavior for each logical display, defined as follows.

- o Synchronous: display requests received for this stream will be performed on the logical display only if the affected part of the logical display is currently windowed. If the request would change an unseen part of the logical display, or if the logical display isn't currently windowed at all, the display request (and all subsequent ones) would be blocked until the affected part of the logical display is visible on the canonical display. The terminal user should be informed that this blocking has happened.

- o Asynchronous: display requests will change the logical display whether it is windowed or not. The terminal user will always be able to see the "current" state of the logical display, but any preceding states will have been seen only if they occurred while that part of the logical display was windowed.

A logical display used to display a time-of-day clock would probably be set up with asynchronous behavior, while a compiler producing error messages might warrant a synchronous display.

Perhaps the program should be able to make this selection in addition to or instead of the terminal user.

6.1.10 Scroll "pad" for page CD

In the special case where a scroll logical display (which is a single-line structure) is being used with a page canonical display, the Window Service will maintain a log of previous logical display images for each such logical display. This log is called a pad; its size will be configurable by the terminal user (within limits imposed by the Terminal Service). When such a logical display is allocated a window, the window will display the current line and as many previous lines of the pad as will fit. The terminal user can scroll the pad behind the window to view previous display images at will (see the "window scrolling" service).

No such log of previous display images will be associated with any page logical display. (Such a service could be provided outside the Terminal Service, if desired.)

6.2 Window interfaces

This section provides an overview of the Window Service's interfaces to the rest of the Terminal Service.

6.2.1 Display/keyboard interfaces

The Window Service reads display requests from all Logical Terminal Servers, as they are available. Each display lexeme is screened and translated into a set of lexemes which would have the appropriate effects on the appropriate window; those lexemes will be forwarded to the Canonical Terminal Server. For example, an "erase display" request for a logical display must be translated into lexemes which will cause the erasure of that part of the canonical display within window boundaries for that logical display (possibly none).

The Window Service reads keyboard lexemes from the Canonical Terminal Server, and routes them to the Logical Terminal Server whose logical keyboard is currently designated the "active" one. There is always an active logical keyboard. The Window Service simply copies input lexemes to the appropriate output path; no translation is required.

The lexeme vocabulary used by the Window Service is the same as that understood by the Canonical Terminal Server. The Window Service must know exactly what effect each lexeme would have on the canonical display, and be prepared to duplicate those effects, subject to window constraints.

Neither the Logical Terminal Server nor the Canonical Terminal Server is aware of the existence of the Window Service.

6.2.2 Managing windows

The terminal user will use a set of interfaces defined by the Human Interface Service to create, delete, move, and change windows, to switch his input to a different logical keyboard, and to examine any relevant information (names, window allocation) about the set of logical terminals associated with his terminal.

Although most programs will be unaware of the existence of windows, we believe that those programs which are prepared to take advantage of the window mechanism (the EMACS screen editor is a good example) should be able to recommend window size and placement on the canonical display. There will have to be some way to resolve conflicts between programs and terminal users as to which windows belong where.

6.3 Other work on Windows

Windows are obviously an idea whose time is coming fast at Prime. In the last year there have been several efforts to specify window operation within the context of specific applications. In addition, there are several ongoing prototyping efforts which have built simple implementations of window mechanisms. The following is a partial list of projects (with documentation, where available) which have thought or are thinking about windows.

- The Unicorn workstation architecture, described in PE-TI-917, "Unicorn Phase I Report", by Jay Goldman, Hugo Strubbe, Doug Voorhies, and Dick Wolfson. A simple implementation, called Minicorn, of the Unicorn's window mechanism is described in PE-TI-978, "Be friendly to the users: try screen-oriented output!", by Hugh Strubbe.

- The beginnings of a proposed Office Automation (3.X) architecture, described in PE-TI-981, "OAS 3.X Architecture Team Interim Report", by Lee Scheffler for the OAS 3.X Architecture Team.

- The Virtual Terminal Project (Research Department) has built a prototype window mechanism, described in PE-TI-944, "Windows with Text or Menus for Three Glass TTY's", by Peter Stein, and PE-TI-945, "A Virtual Terminal System for Fox, Beehive, and HDS Terminals", by Ilya Gertner and Peter Stein.

- The CASE requirements specification includes some terminal user requirements for window operation; see PE-TI-871, "CASE Requirements Specification", by the CASE Development Team.

- The EMACS screen editor is a single process which is capable of dividing the physical display into several independently operating windows.

7 The Logical Terminal Server (Level 3)

Level 3 is the "top" level of the Terminal Service. This level is the closest to the application program, and farthest from the physical terminal. The server at this level is called a the Logical Terminal Server, or LTS; it provides a set of value-added services on top of the standard display and keyboard services available from the server at level 1.

The Logical Terminal Server acts as a filter operating upon the input and output streams which connect the program to the canonical terminal. A client of the LTS invokes output services by sending it standard display lexemes; the LTS will apply some set of transformations to these lexemes, and send the lexemes so produced onward to lower levels of the Terminal Service. The LTS provides input services by applying a different set of transformations to the lexemes it receives from the rest of the Terminal Service, sending the output of that transformation onward to the program. The filtering job done by the LTS may be quite simple (only a few lexemes are altered) or quite complex (incoming and outgoing streams differ radically).

To the client, the LTS appears to modify the behavior of the canonical terminal. It can add a service which the CT doesn't provide directly (as long as the LTS can combine existing CT primitives to get the desired effect). It can appear to change the way in which CT services operate, by slightly changing the set of lexemes sent to or received from the Canonical Terminal Server. This apparently-changed behavior of the CT results in the appearance of what we call a logical terminal.

There are many possible services that can be provided by a level 3 server. The set of services that we will provide is intended to satisfy the requirements stated in PE-TI-844, "Canonical Terminal Requirements". We have been calling this set the "General Purpose Interactive Terminal Services" or GPITS, for short. The name is subject to change without notice! An overview of the services provided within GPITS will be presented later in this section.

GPITS will be the standard terminal support environment available to software builders using PRIME computers. Software builders can replace GPITS by either an extended version of GPITS which contains additional services, or by a completely new package of services intended to provide the appearance of a different terminal support environment and a different logical terminal. Different level 3 servers could supply text editing services (including movement of text blocks within the display) or forms mode services (including definition of protected fields). This possibility is discussed in a later section.

7.1 Invoking the right Level 3 service package

[At LIO open? Specified by the program?]

7.2 The GPITS Services

The primary source for the definition of GPITS services has been the set of requirements presented in the Canonical Terminal Requirements document. We have added a few things (e.g. the "phantom column" service) which we feel are valuable additions to a general terminal service package.

This section provides only an overview of the services GPITS will provide. Detailed specifications of all services, including the interactions among them and the management interfaces used to invoke and control them, will be developed for a separate document, the Terminal Service Functional Specification.

7.2.1 Discard Output

The terminal user can request that the Terminal Service discard all output lexemes. When this service is in effect, all output lexemes from the program and all echoed lexemes generated within GPITS will be discarded within GPITS, without being sent to lower levels of the Terminal Service.

Discarding of output normally happens without the knowledge of the program, and is strictly for the convenience of the terminal user. However, a program will be able to request that particular output messages (important prompts or error messages) be exempt from this service, so that they may be displayed even when the terminal user has requested discarding.

This service is normally turned on and off from the keyboard by the <control-O> lexeme. The program or terminal user can change the lexeme which invokes this service through the Parameter Management Service.

7.2.2 Suspend/Resume Output

The terminal user can request that the Terminal Service suspend or resume processing of output lexemes. When output is suspended, GPITS continues to accept lexemes from the program but will neither operate on them nor send them on to other levels of the Terminal Service. When output is resumed, GPITS will resume normal output processing from the point at which it was suspended. Programs will not be able to override the terminal user's suspend and resume requests.

When this service is in effect, the processing of output lexemes from the program and echoed lexemes generated within GPITS will be suspended.

The terminal user normally requests "suspend output" with <control-S>, and "resume output" with <control-Q>. The program or terminal user can change the lexemes which invoke this service through the Parameter Management Service.

7.2.3 Pagination

The terminal user will be able to define arbitrarily sized collections of "lines", called "pages". When GPITS has sent a page's worth of lines to the display, GPITS will suspend processing on any output received from the program until the terminal user requests the next page.

The pagination service applies only to consecutive lines of output text from the program. Anything echoed by GPITS in response to keyboard input will reset the count.

A "wrapped" output line (see "Line Wrapping Service") will be counted as two or more lines for the purpose of determining end-of-page.

The service will be available only when the program has requested a scroll class display; pagination will not be done when the terminal is being used as a page class display. This service is normally useful only when the physical terminal is a CRT or other device capable of displaying some fixed number of lines at once (such as a letter quality printer which operates on individual sheets of paper.

The page size will normally be chosen by the Terminal Service to correspond to the size of the physical screen or page, or, if the Window Service is active, to the size of the window allocated to this logical terminal. However, the terminal user will be able to set the page count to any other value through the Parameter Management Service.

7.2.4 Attentions

The attention service provides a mechanism for allowing attentions to be invoked by a terminal user. [For now, we are discussing attentions in terms of PRIMOS on_unit invocations; this may not be appropriate in the PDA environment.] The attention service provides:

o [Some undetermined number of] configurable attentions. Each configurable attention is a mapping between a lexeme and an on_unit.

- A mechanism to allow the program to correlate the occurrence of the attention with a position in the input stream. (For instance to allow lexemes typed before the attention to be flushed or otherwise treated differently.)

Programs will be able to enable and disable any and all GPITS attentions.

The attention service by default recognizes one attention. The default attention is invoked by <control-P> and will cause a process's QUIT$ on_unit to be signalled.

The invoked on-unit will be supplied with a terminal identifier and an attention identifier. [Possibly other information.]

7.2.5 Data Forwarding

The data forwarding service allows a program to recommend how the Terminal Service is to block lexemes for transmission over the medium between GPITS and the program. The maximum program responsiveness, every lexeme transmitted to the program as soon as possible, carries the highest transmission cost. The minimum program responsiveness, lexemes grouped together into blocks at the server's discretion, carries the lowest transmission cost. By appropriate use of the mechanism that controls the blocking, programs may trade-off responsiveness for cost.

The default behavior of the data forwarding service is to block lexemes into "lines" (strings of lexemes delimited by a <new line> lexeme). Programs may set up different blocking rules by specifying other lexemes than <new line> to be used as "triggers". A set of trigger lexemes may be designated through the Parameter Management Service.

GPITS may at its discretion, and for any reason, transmit a block of lexemes anytime <u>before</u> a trigger is encountered in the input. The only service <u>guaranteed</u> by data forwarding is that no delay will occur <u>after</u> GPITS encounters a trigger.

7.2.6 Echo

The echo service allows a program to specify the style of echoing which the Terminal Service is to perform on its behalf. Three styles are available.

- Immediate echoing. This is what PRIMOS does today when "full duplex" has been requested. (But see also "Lexeme Mapping", below.)

- No echoing at all. This is what PRIMOS does today when "half duplex" has been requested. The program assumes responsibility for any echoing.

- Deferred echoing. Echoing is performed by the Terminal Service, but can be controlled by the program. in a way that allows orderly formatting of type-ahead. GPITS will suspend echoing when it encounters a "suspender" lexeme, and will resume echoing on command (via the management interface) from the program. The program may designate a set of lexemes to be used as suspenders -- this is normally but not necessarily the same as the set of triggers.

A related service (see Lexeme Mapping) will allow programs to specify a lexeme or string of lexemes to be used when any lexeme is echoed.

Most programs will request immediate echoing, and continue (as today) to have the Terminal Service handle all echoing.

Deferred echoing also makes the Terminal Service handle the work of echoing, but gives the program some control over when and how it's done. It can be used to create the appearance that characters are echoed as they are read by a program, instead of being echoed as they are typed by the terminal user. In a simple example, the <new line> lexeme will be both a suspender and a trigger lexeme. After processing a <new line>, GPITS will suspend echoing of input lexemes until the program tells it to resume echoing. This gives the program a chance to write any output (prompts, etc.) to the display <u>before</u> the typed-ahead characters are echoed. This allows orderly formatting of input and output on the display. The program even has a chance to change the echoing mode (through the management interface) before echoing is resumed. Using this feature, a program can cause a password not to be echoed even if it was typed ahead.

7.2.7 Lexeme Mapping

Lexeme mapping allows a program to specify a mapping or translation from one lexeme to another (or to a string of lexemes). GPITS provides three points at which this mapping takes place.

- Lexemes received from the program are translated according to a <u>display</u> lexeme map before being forwarded to the canonical display.

- Lexemes received from the canonical keyboard are translated according to a <u>input</u> lexeme map before being forwarded to the program.

- Lexemes received from the keyboard are translated according to an echo lexeme map before being echoed to the canonical display.

The three maps may be manipulated separately by the program, through the Parameter Management Service. A default mapping of "lexeme in, lexeme out" is provided.

The lexeme mapping service can be used to make a "soft keyboard"; to change the representation of a lexeme when it's echoed (e.g. to provide control character expansion); or to selectively remove lexemes from the input stream (by making them map into the <nil> lexeme).

7.2.8 Local Editing

The local editing service allows the terminal user to examine and/or make changes in data to be forwarded to the program, without the involvement of the program. The effects of changes to the input data are appropriately reflected on the display, in a manner suitable for the terminal in use (e.g. "back space, blank, back space" on a CRT, or "backslash, erased characters, backslash" on a hard copy terminal). GPITS will determine what is "suitable".

GPITS provides a simple set of editing services:

- erase (removes the single previous lexeme),
- kill (removes all lexemes not yet forwarded to the program), and
- examine (causes the redisplay of all lexemes not yet forwarded to the program).

The GPITS editing service is intended to handle local editing of "lines" -- strings consisting of graphic characters and format effectors, terminated by <new line> or the equivalent. The editing service is closely related to the data forwarding service; only lexemes which GPITS has not yet forwarded -- that is, those processed since the previous "trigger" -- can be edited.

The undoing of display effects will not be guaranteed for certain lexemes (e.g. <position>, <erase>). More sophisticated flavors of local editing may be defined for future versions of GPITS.

The terminal user will normally invoke the three local editing services through a set of lexemes to be specified. The terminal user or a program will be able to designate different lexemes through the Parameter Management Service.

7.2.9 Quoting

The quoting service allows a terminal user to specify that a lexeme from the keyboard is to be sent directly to the program without interpretation by GPITS. It can be used to enter a lexeme as text that would normally be trapped by GPITS as a request for the erase, kill, attention, or other service. Such lexemes will still be subject to the data forwarding, echoing, and lexeme map services.

7.2.10 Variable Tabs

The variable tab service allows a program to specify a set of horizontal tab stops which may be different from the fixed horizontal tab stops used by the canonical display. When this service is in effect, GPITS will convert a <horizontal tab> lexeme into a lexeme string which will cause the canonical display to move the active position to the next program-defined tab stop.

7.2.11 Variable Form Feed Handling

[deleted]

7.2.12 Line Wrapping

When the line wrapping service is in effect, a graphic character displayed in the last column of a line will cause an automatic <new line> to be inserted in the output stream. This will circumvent the normal behavior of the canonical terminal, which is to discard successive characters after an end-of-line boundary violation.

The program or terminal user can turn this service on or off through the Parameter Management Service.

7.2.13 Phantom Column Line Wrapping

Normal line wrapping generates a <new line> after a graphic character causes an end-of-line condition. If the next displayed lexeme is a (real) <new line>, the display will appear to have an "extraneous" blank line.

When the GPITS phantom column service is in effect, the automatic <new line> will be postponed until the next display lexeme is received, and will be omitted entirely if that next display lexeme is a format effector or position lexeme. This will prevent the extraneous blank lines from appearing when the displayed text is exactly as "wide" as the logical [?] display.

7.3 GPITS Interfaces

This section provides an overview of interfaces used by GPITS clients to access GPITS services.

7.3.1 Service Interfaces (Lexemes)

GPITS receives a lexeme at a time from the program output stream. It sends groups of lexemes to the program input stream, where the grouping is determined by the Data Forwarding Service. (One possible grouping is single lexeme at a time.)

The lexeme vocabulary used by GPITS is exactly the same as that of the Basic Canonical Terminal Server (BCTS). All lexemes defined for BCTS in a previous section will be accepted by GPITS for either input or output processing. However, some will be interpreted by GPITS as requests for services and will be removed from the stream between program and canonical terminal. For example, a <horizontal tab> lexeme received by GPITS from either the logical keyboard or the program will be interpreted as a request for the variable tab service. (GPITS may or may not send a <horizontal tab> to the canonical display in performing this service.)

7.3.2 Management Interfaces (parameters)

[List of all parameters and their acceptable values -- to be specified]

7.4 About the GPITS server

A functional design for GPITS will include descriptions of

- o data structures: objects which maintain state, describe services, or otherwise hold information needed by GPITS to perform services;
- o workers: little servers which perform the GPITS services in response to the receipt of display lexemes or during the generation of keyboard lexemes.

[We're not sure that this type of functional design material belongs in this architecture document as it's currently scoped, but here it is anyway.]

7.4.1 Structures

[Internal structures maintained by GPITS -- this section to be specified later.]

7.4.1.1 Toggles to turn services on/off

[Pagination mode, wrap/truncate mode, phantom column mode, echo mode, etc.]

7.4.1.2 Tables to map lexemes to service invocations

[Suspend table, attention table, trigger table, output control table, edit control table, quote table.]

7.4.1.3 Auxiliary program-specified info for services

[Variable tab setting, lexeme maps, attention mode.]

7.4.1.4 Internal structures used by GPITS during services

[Output state, echo state, lexeme buffer(s), data forwarding buffer.]

7.4.2 Workers

We model the GPITS server as consisting of four cooperating workers. (Workers or mini-servers are introduced in the Concepts section.)

GPITS normally deals with four single-directional streams: a pair (one input, one output) which convey data between a program and a "logical terminal" (meaning GPITS), and a pair (one from keyboard, one to display) which convey data between GPITS and the physical display, via the rest of the Terminal Service.

A program may elect to connect only a single input or a single output stream to a logical terminal. Even when this is so, GPITS actively maintains the pair which connect GPITS with the rest of the Terminal Service. This is because several GPITS services are defined to link the logical keyboard with the logical display, independent of any streams between the program and the logical terminal. For instance, the definitions of echoing, local editing, and discard output allow the terminal user to affect the logical display from the logical keyboard. This behavior is guaranteed whether or not there is a stream from the program to the logical display.

```
              |          Lexemes                 ^
              |      From / To Program           |
              |      via Logical I/O             |
              v                                  |
    +---------------------------------------------------+
    |         |                                  ^      |
    |         v                                  |      |
    |   +---------+     +----------+    +------------+  |
    |   | Output  | <-| Deferred  |<-|  Keyboard   |  |
    |   | Control |     | Echo      |   |  Input      |  |
    |   +---------+     +----------+    |  Worker     |  |
    |       |  |                        +------------+  |
    |       |  |                              ^         |
    |       v  v                              |         |
    |  +-----------+                          |         |
    |  | Display   |                          |         |
    |  | Output    |                          |         |
    |  | Worker    |                          |         |
    |  +-----------+                          |         |
    |         |                               |         |
    |         v                               |         |
    +---------------------------------------------------+
              |          Lexemes                ^
              |     From / To Lower Levels      |
              v     of the Terminal Service     |
```

Figure 7.1: Inside the GPITS Level 3 Server

As before, remember that this is not intended as an internal design of the GPITS server. The decomposition of GPITS into workers is just a notational convenience; we found GPITS too complex to describe as a single unit, and easier to think of as asynchronously operating, independent subunits. We could easily have chosen a different decomposition resulting in more or fewer workers with different functions. The only significant feature of any description of GPITS will be its specification of what operations are to be performed on which lexemes and in what order.

There are four workers in GPITS (see Figure 7.1).

7.4.2.1 The Display Output Worker

This worker provides all services which affect the terminal's display, including variable tabs, the displayed effects of local editing, and the phantom column service.

The Display Output Worker receives lexemes from the program output stream and from the internal echo path. It sends completely-processed lexemes to lower levels of the Terminal Service, where they will act on the "private canonical display" associated with this logical terminal.

This worker follows these rules:

- Lexemes from the echo path only are checked to see if they are invocations of the Local Editing service. If so, the worker makes the appropriate changes to the display by generating a set of lexemes which will have the desired effect on the private canonical display.

- Lexemes are run through the appropriate map (there is a different map for program output and echo paths). For example, a <control-C> lexeme may be mapped to the two-lexeme string <^> <C>.

- Miscellaneous display services (line wrapping, variable tab) are applied. Lexemes may be changed or created in the process.

- Finally, the completely-processed lexeme(s) will be sent to the rest of the Terminal Service.

This worker keeps track of anything it will need to perform the guaranteed services. For instance, the current active position on the private canonical display is needed for line wrapping; the active position before a <horizontal tab> was performed may be needed in order to erase that tab.

7.4.2.2 The Output Control Worker

This worker controls the flow of lexemes either from the keyboard (via the echo path) or from the program to the display. It follows these rules:

- If the terminal user has invoked the Suspend service (a.k.a. XOFF), both paths are blocked and nothing goes through until the user Resumes output.

- If the terminal user has invoked the Discard service, lexemes from either path are read and thrown away instead of going to the display.

- If an end-of-page condition (as defined by the Pagination service) is in effect, output from the program is held up until the terminal user gives his O.K. to continue.
- Otherwise, in normal operation, this worker just passes lexemes from either path straight through to the Display Output Worker.

7.4.2.3 The Deferred Echo Worker

This worker provides the (optional) synchronization of echoing with program control. It is essentially a valve which starts and stops the movement of lexemes from the keyboard (via the Keyboard Input Worker) to the display (via the Output Control and Display Output Workers). When normal echoing is in effect, the valve is always open and this worker does nothing but move lexemes along the echo path. When deferred echoing is in effect, this worker closes the valve after a suspend lexeme has gone through, and reopens it when the program so requests.

7.4.2.4 The Keyboard Input Worker

This worker processes lexemes received from the lower levels of the Terminal Service. It sends completely-processed lexemes to the program, blocked according to data forwarding considerations, and sends to-be-echoed lexemes to the Display Output Worker, along the internal echo path.

The worker follows these rules whenever a lexeme arrives:

- If this is a "quoting" lexeme, the next lexeme will be treated as text and will not be examined for service invocations.
- If this is an "attention" lexeme, the appropriate attention will be signalled for the program. The lexeme may be left in the stream so that all input before the attention can be easily identified (e.g. to be flushed) by the program.
- If this is an "output control" lexeme (invokes the Suspend, Resume, Discard service), the Output Control Worker must be informed of a change in state.
- If this is a "local edit" lexeme, the appropriate editing service should be performed on any lexemes not yet forwarded to the program.

- Otherwise, for normal lexemes, the worker will send a copy of the lexeme to the echo path, apply the lexeme mapping service, and put the resulting lexeme(s) into a data forwarding buffer.

- Whenever a "trigger" lexeme has been buffered, the worker will send all accumulated lexemes to the program.

[The STROMA descriptions of worker algorithms, which were present in the first draft, have been moved to a separate document.]

7.4.3 Notes to the Implementor

(not part of the architectural specification)

This section provides guidelines and suggestions to Terminal Service implementors about how the services defined by GPITS might be provided in various terminal support configurations.

[Might distribute GPITS work among several components for something small like FALCON ... local editing effects may be different for different underlying terminals (hardcopy vs. screen) ...]

8 The Parameter Management Service

[This section is very preliminary and will be filled in later.]

The Parameter Mangement Service provide programs with two services:

- o An ability to manipulate parameters within a logical terminal, i.e., within a stream, controlling the behaviour of that logical terminal.
- o Logical Terminal structure independence, i.e., parameters appear to belong to the logical terminal as a whole (a program need not know what components of the Terminal Service actually use those parameters.

Parameters have special characteristics. These are:

- o Parameters and structures are not the same things. Structures are implementation dependent, parameters are architectural concepts. Structures will be visible or interesting only within the Terminal Service; parameters are externally visible.
- o Parameters have unique names, independent of the implementation of terminal service.
- o Each parameter has an owner. Only the owner of a parameter may manipulate the value of a parameter.

8.1 The Parameter Management Servers

Early on, we realized that parameters were associated with particular portions of the terminal service. For example, if a parameter exists that defines the size of a window for a stream, that parameter is most likely associated with the window services.

This type of association argued in favor of particular service owning particular parameters. At the same time, we felt that the program should not have to know that parameters were associated with particular services, i.e., we wanted to present the impression of a "flat" name space for parameters. The structure that we have defined combines both of our desires for parameter mangement:

- o Parameters "owned" by only server and
- o a flat name space for programs manipulating parameters.

```
+----------------------------------------------------+
|                      Program                       |
+----------------------------------------------------+
                         |                        |
                         |                        |
                         |                        |
                         |             +---------------------+
                         |             | Service to manage   |
                         |             | parameters          |
                         |             | controlling ...     |
 +----------------------------+        |---------------------|
 |                            |-|      | Logical Terminal    |
 +----------------------------+        | Services            |
                         |             |---------------------|
                         |                        |
                         |                        |
       +------------------+            |---------------------|
       |                  |------------| Window Services     |
       +------------------+            |---------------------|
                         |                        |
                         |                        |
+-----------------------------+        |---------------------|
|                             |-|      | Canonical Terminal  |
+-----------------------------+        | Services            |
                         |             +---------------------+
                         |
                         |
      +-------------------+
      |                   |
      +-------------------+
                |
            Terminal
            Operator
```

Figure 8.1: Inside the Parameter Management Service

8.1.1 Lexemes for Parameter Management

[We anticipate a parameter set/read protocol which is completely independent of the transfer of data lexemes. We would like to see parameter names defined in common for all PDA services, not just the Terminal Service, at least for common items.]

- <set {parameter-name} {parameter-value}>
  Specifies a parameter name and a value for that parameter. The format of the parameter value will vary depending on the data type of the parameter name being specified.

- <read {parameter-name}>
  Requests the value of the specified parameter. The format of the parameter value will vary depending on the data type of the parameter.

- <response {parameter-name} {parameter-value}>
  The value of the parameter specified by a <read> lexeme.

8.1.2 Protocol for Parameter Management

[to be supplied]

8.1.3 Notes to the Implementor

[probably need examples of the three types of parameter management interaction: from program, management component, logical terminal component].

9 The Terminal Service Overseer

[This section is very preliminary and will be filled in later.]

[Managing streams in the PDA environment is not well understood. Thus this section will not discuss how services are achieved. We will talk about the types of services that must be provided in order to make things work. We expect much of the requirements definition of these services to come from two sources:

o The functional design of the logical terminal.

o The functional design of the Human Interface Services.]

The Terminal Service Overseer serves two distinct clients: other components inside the Terminal Service and clients outside the Terminal Service. Services provided to clients within the Terminal Service tend to be interfaces to the environment outside the Terminal Service. Services provided to clients outside the Terminal Service tend to manipulate streams.

9.1 Services

9.1.1 Initialization Services

There are two types of initialization service:

o Canonical Terminal Initialization and

o Logical Terminal (or Stream) Initialization

CT Initialzation deals with selecting the correct set of CT Services (for example what kind of CT(s) to provide? Basic, Graphics, or Forms CT), connecting a CT to the correct window services, etc.

LT Initialization allowis Logical I/O to select a logical terminal service package, connecting a terminal user to an appropriate human interface and job management service, resource allocation for data bases, etc.

9.1.2 Security Service

Part of the reason for the Terminal Service Overseer is the need for the Human Interface Service (and potentially other services) to control the state of "other" logical terminals. To keep this capability from being mis-used the Terminal Service will require some kind of validation before allowing the use of the more potentially destructive services.

9.1.3 Stream Manipulation Services

There are probably two types of services related to stream manipulation.

- o Stream Status Services
- o Stream Control Services

Stream Status Services would allow outside clients to request information about streams, how many streams does a user have, what are they connected to, how many I/O operations have been performed, etc.

Stream Control Services would allow outside clients to modify the state of streams that are (1) possibly not owned by the client and (2) certainly not the normally referenced stream (in the case of the Human Interface Service managing parameters, stream control services would have to be invoked to change parameters in "someone else's" or "some other" stream).

9.1.4 Attention Delivery Service

Allows the Terminal Service to deliver attentions (at least QUIT$) to processes in the "outside" environment.

9.1.5 Process Status Service

Allows the Terminal Service to request information about the outside environment. [For those people who are familiar with DECsystem-10/20s we are talking about <control-T>.]

9.2 The Terminal Service Overseer

9.3 Notes on Implementation

- o The Terminal Service Overseer provide the Terminal Service with environmental isolation. We can't say much about the needs of the Terminal Service (we haven't done a functional design), we can't say much about the needs of clients outside the Terminal Service (we haven't designed a human interface) but the Terminal Server Overseer is where the two, possibly conflicting, set of needs meet.

10 The Human Interface Service

[We have done no work in this area. We expect requirements for this service to come from the group specifying user interfaces in a PDA environment.]

11 Proposal for further application of the architecture

One of the unsatisfying things about this document is the lack of an overall structure that shows how all types of terminals, forms, basic, and graphics, are supported by the Terminal Service. This section discusses the larger view of the Terminal Service.

In this section, we start with the assertion that the canonical terminal level of the Terminal Service Architecture describes a model of the appropriate type of terminal. The only canonical terminals we will define are:

- o The Forms Canonical Terminal.
  [fields as primitive units]

- o The Basic Canonical Terminal.
  [described at length elsewhere in this document]. [cells as primitive units]

- o The Graphics Canonical Terminal.
  [pixels as primitive units.]

The Window Server understands the structure of each type of canonical display; [whether there is only one window server that knows about every possible CD structure or a window server per CD structure is a anyone's guess right now.] the kind of primitive units it's made of, the size of the array of primitive units, and the rules which the canonical display uses to manipulate the primitive units. The Window Server takes lexemes for operations describing operations on the canonical display and truncates or transforms them according to the window bounds. The lexemes received by the Window Server always deal with the primitive unit of the canonical display, e.g., the window server never sees forms canonical terminal lexemes intended for a graphics canonical terminal. In other words, the Window Server handling a graphics canonical terminal doesn't know about cells (primitive unit of the basic canonical terminal), and the Window Server handling a basic canonical terminal doesn't know about fields (primitive unit of the forms canonical terminal).

Obviously a canonical terminal whose primitive units are pixels can be used to display the "larger" constructs which are cells (aggregates of pixels) and fields (aggregates of cells). But the emulation of these larger primitive units is only possible if an architectural component of the Terminal Service Architecture above the Window Server is willing to convert cell or field lexemes into pixel lexemes. A more subtle point, a canonical terminal whose primitive units are fields cannot display anything "smaller" than fields; cells and pixels cannot exist on a forms canonical terminal.

Lets look at a real example. Suppose the physical terminal is capable of supporting field operations in "forms" mode and cell operations in "character-at-a-time" mode (the OWL, PT45, and WREN all provide both capabilities.) Then the implementor of the canonical terminal "drivers" has a choice when providing forms support for these terminals.

- The implementor can choose to implement a Forms Canonical Terminal. A forms CT takes advantage of the block and field capabilities of the terminal and gains substantial performance improvements. However, mapping canonical fields onto the fields understood by the physical terminal may be difficult and a forms canonical terminal cannot provide basic canonical terminal-style access to the physical terminal while the physical terminal is a forms canonical terminal, e.g., no windows to the PDA command processor.

- The implementor can choose to implement a Basic Canonical Terminal, putting the Forms canonical terminal functions in a separate field-cell conversion layer (see the "Basic Canonical Terminal" picture which follows). This approach gains flexibility; any kind of field can be fairly easily emulated with cells and the terminal user has access to different kinds of windows (scroll, page and forms) all at the same time. An added attraction for the terminal user is that Forms Canonical Terminal functionality is available when the Basic Canonical Terminal is supporting a FOX, DM10, ADM3A, or other dumb terminal. The terminal user pays for this kind of flexibility by losing the ability to use those features of the OWL, PT45, WREN, or whatever, which could save CPU cylces and/or line transmission time. Terminals like the OWL, PT45, and WREN typically cost a little more than dumb terminals so the terminal user looses a little money also.

  [Existence of cheap terminal controllers or concentrators that provide the Terminal Service may change the cost picture.]

[We need to understand how swapping between Forms and Basic canonical terminals works when the terminal (e.g. WREN) is being used first for one, then for the other.]

The several pages show pictures of the Terminal Service Architecture (without the Management Server and Human Interface Service) with the our proposals (and alternates) for handling the three kinds of canonical displays: Forms, Basic, and Graphics.

```
                         To/from application
               |                                  |
               |                                  |
     +--------------------+           +--------------------+
     | FORMS Logical      |           | FORMS Logical      |
     | Terminal Services  |           | Terminal Services  |
     +--------------------+           +--------------------+
               ^                                  ^
               |     <-- field lexemes -->        |
               |                                  |
               v                                  v
              +-----------------------------+
              |   Field Window Services     |
              +-----------------------------+
                           ^
                           |
                           |   <-- field lexemes
                           v
                  +-------------------+
                  | FORMS Canonical   |
                  | Terminal Services |
                  +-------------------+
```

Figure 11.1: Terminal Service Architecture with Forms Canonical Terminal

Notes:

o The Forms CT is block mode only. You send it blocks of field lexemes and it will return the same. The "turn" concept is probably built in.

o The Forms CT deals only in "whole" fields, not "partial" ones. The windows must contain entire fields, they can't be split. The "logical display" seen from level 3 and up may be larger than the window as long as this restriction is obeyed. (You would probably have top and bottom windows rather than side by side ones.)

o Auto tabbing, a relationship between fields, may need to involve window services.

o Scroll/page behavior is <u>not</u> supported on the Forms CT; there can be no page windows mixed in with the forms windows. (Of course this behavior may be emulated by an application package above the logical terminal services which works like DPTX/TSF's "data handler".)

o It's possible we nay want different definitions of the Forms CT, e.g. for TP vs. DPTX, or for the Forms Logical Terminal Services.

```
                  To/from application
         |                                |
         |                                |
   +-------------------+                  |
   | FORMS Logical     |                  |
   | Terminal Services |                  |
   +-------------------+                  |
         ^                                |
         |  <-- field lexemes             |  <-- cell lexemes
         |                                |
         v                                |
    +----------------+          +-------------------+
    |  field-cell    |          | Basic Logical     |
    |  converter     |          | Terminal Services |
    +----------------+          | (GPITS)           |
         ^                      +-------------------+
         |                                ^
         |  <-- cell lexemes -->          |
         |                                |
         v                                v
          +-----------------------------+
          |    Cell Window Services     |
          +-----------------------------+
                       ^
                       |
                       |  <-- cell lexemes
                       v
              +-------------------+
              | Basic Canonical   |
              | Terminal Services |
              +-------------------+
```

Figure 11.2: Terminal Service Architecture with Basic Canonical Terminal

Notes:

- The "field-cell" converter gets a character at a time in from the Basic CT and performs all echo, edit, etc. behavior associated with the Forms CT.

- It's now possible to have field and cell windows mixed. It's also possible to have an incomplete field in a window as long as it contains complete cells! The field-cell converter will turn a field display request into a string of cell display requests; the Window Server will just truncate to the appropriate cell.

```
                          To/from application
     |                          |                           |
     |                          |                           |
+--------------------+          |                           |
| FORMS Logical      |          |                           |
| Terminal Services  |          |                           |
+--------------------+          |                           |
     ^                          |                           |
     | <-- field                | <-- cell                  | <-- pixel
     |     lexemes              |     lexemes               |     lexemes
     v                          |                           |
 +-----------------+       +--------------------+           |
 |  field-cell     |       | Basic Logical      |           |
 |  converter      |       | Terminal Services  |           |
 +-----------------+       | (GPITS)            |           |
     ^                     +--------------------+           |
     |                          ^                           |
     |     <-- cell lexemes --> |                           |
     v                          v                           |
 +-----------------+        +-----------------+       +--------------------+
 |  pixel-cell     |        |  pixel-cell     |       | Graphics Logical   |
 |  converter      |        |  converter      |       | Teminal Services   |
 +-----------------+        +-----------------+       +--------------------+
     ^                          ^                           ^
     |               <-- pixel  |  lexemes -->              |
     v                          v                           v
        +-------------------------------------------------------+
        |                  Pixel Window Services                |
        +-------------------------------------------------------+
                                ^
                                |  <-- pixel lexemes
                                v
                           +---------------------+
                           | Graphics Canonical  |
                           | Terminal Services   |
                           +---------------------+
```

Figure 11.3: Terminal Service Architecture with Graphics Canonical Terminal

Notes:

o Don't know much about graphics primitives; we're relying on analogy with forms/basic, which we do understand, to support this model of basic/graphics.

```
                      To/from application

          |                      |                      |
+-------------------+  +-------------------+  +-------------------+
| FORMS Logical     |  | Basic Logical     |  | Graphics Logical  |
| Terminal Services |  | Terminal Services |  | Terminal Services |
+-------------------+  +-------------------+  +-------------------+
          ^                      ^                       ^
          | <-- field            | <-- cell              | <-- pixel
          |     lexemes          |     lexemes           |     lexemes
          v                      v                       v
+-------------------+  +-------------------+  +-------------------+
|       Field       |  |       Cell        |  |       Pixel       |
|  Window Services  |  |  Window Services  |  |  Window Services  |
+-------------------+  +-------------------+  +-------------------+
          ^                      ^                       ^
          |                      |                       |
          v                      |                       |
   +------------+                |                       |
   | field-cell |                |                       |
   | converter  |                |                       |
   +------------+                |                       |
          ^                      |                       |
          | <-- cell lexemes     |                       |
          v                      |                       |
   +------------+         +------------+                 |
   | cell-pixel |         | cell-pixel |                 |
   | converter  |         | converter  |                 |
   +------------+         +------------+                 |
          ^                      ^                       |
          |            <-- pixel | lexemes -->           |
          |                      |                       |
          +---------+            |     +-----------------+
                    v            v     v
                   +--------------------+
                   | Graphics Canonical |
                   | Terminal Services  |
                   +--------------------+
```

Figure 11.4: Terminal Service Archtecture with Multiple Canonical Terminals

o Note that another possibile place for the converters does exist -- "below" the window services. In this configuration, the canonical terminal services plus converters are other forms of canonical terminal services, e.g., a graphics CT with a cell-pixel is a basic CT (note also that most real graphics terminals are page mode terminals also). Similar analogies can be made for other types of converters and CT's.

The work that remains to be done is:

- o The Window Server needs to be fully understood for all types of canonical terminal.
- o The Forms and Graphics Canonical Terminal need to be defined. [We are looking the the UK Forms Management Terminal Service project to define the Forms CT. Perhaps the CAD/CAM or UNICORN projects should define the Graphics CT?]
- o The converters need to be constructed.

Once this work is done, the next step is to start building hardware that directly supports arbitrary level 3 service packages, converters, window management, and canonical terminals. [WREN 2, UNICORN or the CAD/CAM workstation, BEAVER, HAWK/FALCON, EAGLE, etc. are all candidates that might directly support some or all of the Terminal Service.]

12 Glossary

[to be supplied]