Pads Programming Guide – Revision 1

ABSTRACT

`Pads` is an object-oriented user interface package for bitmap terminals, most suitable for browser-like applications. Written in C++ and linked with the `Pads` library, an application program instantiates objects and describes them to `Pads`, creating windows and lines of text, to which the objects are bound. Asynchronously, `Pads` manages the display of this set of overlapping, scrolling windows and maps each semantic operation from the user to a call of a member function of the appropriate object.

In the current implementation the application is a timesharing process on a UNIX† system, communicating with a generic terminal process in a DMD 5620 or Blit terminal. The abstractions used in the `Pads` interface make few assumptions about the terminal's properties and it should be possible to use `Pads` in a variety of other environments.

This document contains: (i) an explanation of `Pads`' non-sequential programming model, (ii) a description of the mechanics of the 5620/Blit human interface, (iii) some sample programs, (iv) the details of programming with `Pads`, which assumes some familiarity with C++.

1. Programming Model

`Pads` is a bitmap user interface package based on an object-oriented programming model. An *object* in `Pads` is the instantiation of a `C++`[1] class: a collection of data with a set of operations. The human user is presented with textual descriptions of objects, each of which is augmented with a set of operations. When the user selects an object on the screen, its set of operations becomes available in a menu. If one of the menu entries is selected, its operation is invoked on the object. A line of input from the keyboard may also be directed to an object.

The global control flow in such a program is unlike the sequential flow of a conventional procedure-oriented program. The execution of a `Pads` program is driven by direct invocations of its objects' member functions. The application has no global control for interpreting an input stream; member functions are invoked from an external control mechanism. Usually each invocation results in a change to the display, showing the user some new or changed information. Each function invocation is completed in isolation and must leave the program ready to accept the user's next operation.

Compared to procedural software, more of the program is devoted to deciding what set of applicable operations should be presented to the user and less to trying to guide the user and interpret input. Error handling techniques illustrate the difference. Suppose a program finds itself in the middle of a complex operation when it cannot open a file. In the procedural model, the program might:

(a) issue an error message and abort the operation. The user must fix the problem and restart the the operation, possibly having to repeat a lot of work to recover the lost context.

(b) issue an error message and pause with a prompt that lets the user choose whether to try again or give up. This at least gives the user a chance to continue. But the program itself cannot help to fix the problem; it is stuck waiting for an answer to the question.

In `Pads`, when an object runs into trouble, it can report the problem and place a set of recovery actions in its menu. It does nothing more until the user chooses to send it another operation. The user can

† UNIX is a trademark of AT&T Bell Laboratories.

communicate with other objects to fix the problem, returning to it arbitrarily later, or ignoring it forever. Effectively, an object *cannot* pose a question to the user and demand a response before the program as a whole proceeds. To do so it would have to arrange that all other objects remove their menus, leaving the user no choice. Rather, it must leave itself in a state from which it can continue, at the user's behest.

Different implementations of this model are possible. `Pads` runs the application as a host process, communicating with a generic process in the terminal. The host creates objects and binds them to windows or lines of text. For each object a textual description and set of operations is sent to the terminal. The terminal program presents them to the user as a set of overlapping, scrolling windows. The user selects a current window, or a line within a window, and makes menu selections. Each menu selection results in a message to the host that invokes the corresponding member function of the object bound to the window or line.

Host-terminal communications are asynchronous; there is no notion of a reply, result or acknowledgement. The host may send object descriptions to the terminal arbitrarily; the terminal may send operations to the host arbitrarily. Only the order of messages along each path is guaranteed.

2. Human Interaction

The remainder of this document assumes the terminal to be a DMD 5620 or Blit. These are the only terminals for which `Pads` has been implemented to date. Other terminals, say with a one-button mouse, would behave differently.

The application's objects are displayed as a set of windows containing lines of text. The mouse controls window positioning, overlap and scrolling and selects a current window and line. Operations are directed to the current window or line from pop-up menus or the keyboard.

The left mouse button points. Pointing at a window makes it current, with a heavy border; pointing at a line makes it current, video-inverted. The black scroll bar at the left of each window shows proportionally how much of the text of a window is visible; pointing into the scroll region repositions it. To provide finer scrolling in large files, selecting the top or bottom line moves it to the middle of the window.

The middle mouse button menu has line-level operations; they apply to the current line. It is in two parts. The operations above the separator are specific to each line. The operations below the separator are generic line operations:

> **cut** removes the line; **sever** removes the line and all lines above it; **fold** folds a line that is wider than its window, so all of it appears; **truncate** truncates a wide line at the right, so it occupies only one physical line.

The right mouse button menu has window-level operations. It is in three parts. The operations above the upper separator are specific to each window. Below the lower separator is a list of all the available windows; selecting one makes it current. They appear in front-to-back screen order, current at the top. The operations between the separators are generic window operations:

> **reshape, move, close** are familiar; **fold** and **truncate** apply to all the lines in the window.

Some windows and lines accept input from the keyboard. Keyboard characters accumulate at the bottom of the layer. If the current line accepts input, it flashes with each keystroke; otherwise, if the current window accepts input, its border flashes. If neither flashes, carriage return is ignored until a line or window that does flash is selected. On carriage return, the whole input line is sent to the line or window. Hitting backspace to see what flashes is a safe way to find out where an input line would go. Each line or window that accepts keyboard input produces a one-line help message in response to `?`.

The arrow-dot-dot-dot cursor icon indicates that the host is still completing an operation; the terminal is ready *asynchronously*. The exclamation mark icon calls for confirmation of a menu selection by pressing that menu's button again.

3. Sample programs

The null `Pads` program is:

```
#include <pads.pub>

void main()
{
    PadsInit();
    PadsServe();
}
```

It is compiled with the header file `/usr/jerq/include/pads.pub` and the library `/usr/jerq/lib/libpads.a`. These pathnames may vary from system to system. The host code does not depend on the type of terminal; the same files are used for 256K and 1M DMD 5620 and 68000 Blit. The program is compiled:

```
CC null.c -I/usr/jerq/include /usr/jerq/lib/libpads.a
```

When the program is run, `PadsInit()` downloads the terminal program. It must be the first call to `Pads`. Depending on the `TERM` environment variable the download is

```
32ld /usr/jerq/mbin/pads.m
```

or

```
68ld /usr/blit/mbin/pads.m
```

`PadsServe()` hands control of the host process to `Pads`. It never returns; from that point the global control flow of the program remains with `Pads`. The application code is now passive, invoked by calls from `Pads`. `PadsServe()` reads requests from the terminal to invoke operations on objects defined by the application. In this program there are none; nothing more can happen. The host process blocks reading from the terminal in `PadsServe()`.

To produce output, the program must create at least one pad (window). To receive input, a pad must be bound to an instantiation of one of the program's classes, and a menu of operations must be supplied.

The following program creates a window that displays the date when `'date'` is selected from its only pad's menu:

```
#include <pads.pub>
#include <CC/stdio.h>

class Date : public PadRcv {
    class Pad    *pad;
    void     date();
             Date();
};

Date.Date()
{
    Menu m;
    pad = new Pad( (PadRcv*) this );
    pad->banner( "Current Date" );
    pad->name( "date" );
    m.first( "date", (Action) &date );
    pad->menu( m );
    pad->makecurrent();
}

void Date.date()
{
    char *ctime(long*);
    long time(long*), t;
    time(&t);
    pad->insert( 1, "%s", ctime(&t) );
}

void main()
{
    char *error = PadsInit();
    if( error ){
        fprintf( stderr, "%s", error );
        exit(1);
    }
    new Date;
    PadsServe();
}
```

After downloading and checking for an error return from `PadsInit()`, `main()` constructs an instance of `Date`; `Date.Date()` in turn constructs a `Pad`:

```
class Pad {
public:
            Pad(PadRcv *);
    void    banner(char* ...);
    void    name(char* ...);
    void    insert(long, Attrib, PadRcv*, Menu&, char* ...);
    void    insert(long, Attrib, char* ...);
    void    insert(long, char* ...);
    void    makecurrent();
    void    menu(Menu&);
};
```

Definitions like this from `pads.pub` are shown in boxes, and for simplicity may be incomplete. Check `pads.pub` for complete definitions.

In order to pass itself as the argument to `Pad.Pad()` and become the recipient of operations from the pad's window in the terminal, `Date` is derived from `PadRcv`, a `Pads` base class. The definition of `PadRcv` is not needed yet. After the construction, `pad` points to an embryonic pad: empty, anonymous and impotent. The pointer is, of course, to an object in the host process, an agent representing an object in the terminal.

Several different functions are called to establish the pad's properties. `Pad.banner()` takes `printf` arguments producing a string, `'Current Date'`, to appear at the top of the pad's window. Similarly, `Pad.name()` defines a string, `'date'`, by which the pad is to be known in the list of pads at the bottom of the button 3 menu.

Menus are built using the `Pads` class `Menu`:

```
typedef void (*Action) (...);

class Menu {
public:
            Menu();
            ~Menu();
            Menu(  char*, Action, long = 0 );
    void    first( char*, Action, long = 0 );
    void    last(  char*, Action, long = 0 );
    void    sort(  char*, Action, long = 0 );
};
```

The void constructor makes an empty menu. The functions in `Menu` add items into the menu. `Menu.first()` adds an item to the top of the menu. In this example `Menu.first()` is called only once, resulting in a one-element menu. The string, `'date'`, is to be presented to the user, if the menu is raised. The function `date()`, i.e. `Date.date()`, is to be called if the item is selected from the menu. The third (optional) argument of `Menu.first()` is a 32-bit argument to be passed along when the function is called. `Menu.last()` adds to the bottom of the menu. `Menu.sort()` insertion sorts a single entry into the menu. (It doesn't sort the whole menu.) `Pad.menu()` binds a menu to a pad; the menu appears as the top segment of the menu raised by button 3, when the pad is selected. The pointer passed to `Pad`'s constructor determines on which instance of the class `Date` the function `Date.date()` is invoked.

The last statement in `Date.Date()` calls `Pad.makecurrent()` to force the pad to be made visible and selected as current. If hidden, this bring a pad to the ''front'' of the screen; if not displayed at all – the initial state of a pad – this requests the user to sweep a rectangle for it.

After this initialization `main()` calls `PadsServe()`, but nothing will happen until `'date'` is selected from the menu. Then, out of the blue, `Date.date()` is called to determine the time and write it to the pad. The pad has the default form: a sequence of lines ordered by a 32-bit key associated with each inserted line of text. Key `1` is the top line; key `2**32-1` is the bottom. In `Date.date()`

```
pad->insert( 1, "%s", ctime(&t) );
```

inserts the string constructed from the `printf` arguments as the top (and only) line in the pad. If `date()` is re-invoked, it inserts again with the same key and overwrites the previous date.

The next program is a toy news browser. It uses an unbounded number of pads to present the contents of `/usr/news`. Its first pad displays a list of file names from `/usr/news`, one per line. Selecting a line and choosing `'open'` from its menu creates a new pad, showing the file's contents.

```
#include <pads.pub>
#include <CC/stdio.h>

class Story : public PadRcv {
    char file[16];
    FILE *fp;
    long lines;
    Pad *pad;
    void linereq(long,Attrib);
public:
    void open();
        Story(char *f)   { strncpy( file, f, strlen(f)-1 ); }
};

class News  : public PadRcv {
    Pad *pad;
public:
        News();
};

void main()
{
    if( chdir( "/usr/news" ) ) exit(1);
    char *error = PadsInit();
    if( error ){
        fprintf( stderr, "%s", error );
        exit(1);
    }
    new News();
    PadsServe();
}

News.News()
{
    Story *s;
    FILE *fp, *popen(char*,char*);
    Menu m( "open", (Action) &Story::open );
    char file[16];
    long uniq = 0;
    pad = new Pad( (PadRcv *) this );
    pad->options(TRUNCATE|SORTED);
    pad->banner( "News:" );
    pad->name( "News" );
    pad->makecurrent();
    if( !(fp = popen("ls", "r")) ){
        pad->insert( 1, "can't ls" );
        return;
    }
    while( fgets(file, 16, fp) ){
        s = new Story( file );
        pad->insert( ++uniq, (Attrib) 0, (PadRcv*) s, m, "%s", file );
    }
    pclose(fp);
}
```

```
void Story.open()
{
    char buf[256];

    if( !pad ){
        pad = new Pad( (PadRcv*) this );
        pad->banner( "%s:", file );
        pad->name( file );
        if( !(fp = fopen( file, "r" )) ){
            pad->insert( 1, "cannot open file" );
            return;
        }
        lines = 0;
        while( fgets( buf, 256, fp ) ) ++lines;
        pad->lines(lines);
    }
    pad->makecurrent();
}

void Story.linereq( long i, Attrib a )
{
    char buf[256];
    long n;

    fseek( fp, 0, 0 );
    for( n = i; n > 0; --n )
        fgets( buf, 256, fp );
    pad->insert( i, a, "%s ", buf );
}
```

`News.News()` calls `Pad.options()` to change the properties of its pad. `TRUNCATE` specifies that lines in the pad be truncated at the right when their length exceeds the current width of the pad. `SORTED` specifies that the lines in the pad be ordered by dictionary order rather than by key. `TRUNCATE` and `SORTED` are constants of type `Attrib`.

For each file in `/usr/news`, `News.News()` creates a `Story` and inserts the name of the file into the pad. When the `Story` is created, `Story.Story()` merely copies the name of the file to a private area. The call to `Pad.insert()` accomplishes quite a lot:

```
pad->insert( ++uniq, (Attrib) 0, (PadRcv*) s, m, "%s", file );
```

`++uniq` generates an increasing sequence of unique keys. Since the pad is `SORTED`, it doesn't matter what keys are used, but they must differ to prevent an earlier story being overwritten. The `Attrib` argument is the set of attributes to be set for the line – none in this case. The `PadRcv*` and `Menu&` arguments bind an object and menu to the line. In this case a singleton menu is constructed by giving arguments directly to `Menu.Menu()`. Nothing distinguishes the objects and menus that are bound to lines from those that are bound to pads. The `printf` arguments (necessarily at the end) give the text for the line.

When the `'open'` operation is selected on one of these lines, `Story.open()` is called on the corresponding `Story`. For a given `Story`, only the first call to `open()` is interesting. If `open()` has been called previously, it merely makes its pad current. On the first call it creates a pad, reads its file and arranges that the text of the file be sent to the terminal, on demand. By creating the pad whether the file is successfully opened or not, it has somewhere to write an error message if the open fails. If the file is opened successfully, the length is counted in `lines` and passed to `Pad.lines()`.

`Pad.lines()` radically changes the nature of the pad. Instead of waiting for lines of text to be inserted by the host, the terminal now knows the exact number of lines in the pad and actively requests the lines it needs from the pad's object. As the user scrolls the window and the terminal needs to display a line that is doesn't have, a request for the line is sent to the pad's object in the host. To receive the request the host object must define `linereq()`, one of the virtual functions in `class PadRcv`:

```
class PadRcv {
public:
virtual void    cycle();
virtual void    help();
virtual void    kbd(char*);
virtual void    linereq(long,Attrib=0);
virtual void    userclose();
virtual void    usercut();
};
```

`Story.linereq()` will not be called until after the user sweeps out a rectangle in response to the call to `Pad.makecurrent()`. If the rectangle is big enough to hold 3 lines, say, then there will be three calls to `linereq()`:

```
Story.linereq( 1, (Attrib) 0 );
Story.linereq( 2, (Attrib) 0 );
Story.linereq( 3, (Attrib) 0 );
```

Host code should make no assumptions about the sequence of calls it will receive from the the terminal. In particular, the same line may be requested more than once. The terminal caches only a bounded number of lines for a given pad and may re-request those it has discarded. (However, the implementation of the `Pads` host-terminal protocol is optimized for the most common pattern, a sequential run of line numbers – as this program ought to be!)

`linereq()`'s `Attrib` argument will always be `0` on calls from the terminal. The argument is provided so that calls from the host may specify attributes for the inserted line. Some of the attributes for lines are:

```
SELECTLINE
FOLD
TRUNCATE
```

`FOLD` and `TRUNCATE` specify whether a line should be truncated, overriding the pad's attribute. `SELECTLINE` causes the line to be selected as the current line, as though the user had pointed to it. To see how this can interact with `linereq()`, consider adding a (silly) command to display the middle of a news story. A function `middle()` is added to `Story`: **/n/seki/usr/tac/pads/doc/story.c** and a menu is added to the pad in `Story.open()`: **/n/seki/usr/tac/pads/doc/storyopen.c** Now if `'middle'` is selected, `middle()` is called. For a 30-line story, say, `middle()` calls

```
linereq( 15, SELECTLINE )
```

The `SELECTLINE` attribute requests the terminal to select the line, as though the user had pointed to it; and therefore forces the terminal to display the line. Without this the line might not be displayed unless the user scrolls to see it. In fact, the terminal displays a `SELECTLINE` line as close to the physical center of the window as possible. If the surrounding lines are needed, `linereq()` requests will be sent to the host to fetch them. Host code cannot tell the difference between a `linereq()` generated by the user scrolling the pad and a `linereq()` generated by other host code. Notice that `middle()` doesn't need to know anything about the size of the window to get a specific context displayed.

4. The PadRcv Base Class

Skip this section on first reading.

All host objects that are to receive operations from the terminal must be instantiations of some application-defined class derived from `class PadRcv`:

```
class Any : public PadRcv {
    ...
};
```

```
class PadRcv {
       OBJ_ID obj_id;
public:
              PadRcv();
              ~PadRcv();
        void  invalidate();
};
```

`PadRcv.obj_id` guarantees that operations from the terminal aren't applied to deleted host objects. An object is bound to a pad or line by both its address and its unique `obj_id`; an operation from the terminal is applied only if the identifiers agree. An in-flight operation may be directed to an object that is concurrently removing its pad/line and deleting itself and won't exist when the packet arrives. When deleted, `PadRcv.~PadRcv()` changes `obj_id` so that the operation is ignored.

The object may explicitly inhibit operations from all the pads and lines to which it is currently bound by calling `invalidate()`.

```
    class PadRcv {
public:
virtual char    *kbd(char*)   { return "sorry - no kbd"; }
virtual char    *help()       { return "sorry - no help"; }
};
```

If attribute `ACCEPT_KBD` is specified for a pad or line, `kbd()` is called when the user selects the pad or line and enters a line of text from the keyboard. There is no `\n` or `\r` at the end of the string. A non-zero return from `kbd()` is an error message for the user. `ACCEPT_KBD` is set implicitly by `Pad.Pad(obj)`, if `obj` defines `kbd()`, but must be set explicitly for lines:

```
pad->insert(key, ACCEPT_KBD ...);
```

If the keyboard string is exactly `'?'`, `help()` is called instead of `kbd()`. `help()` should return a string to be presented to the user in a single line.

```
    class PadRcv {
public:
virtual void   userclose();
virtual void   usercut();
};
```

When the user closes a window or cuts a line on the screen, the host object may be informed. If the attribute `USERCLOSE(USERCUT)` is specified for the pad(line), `userclose(usercut)` is called for the host object when the user closes(cuts). All record of the pad(line) in the terminal is removed unless the `DONT_CLOSE(DONT_CUT)` attribute is set. `USERCLOSE` is set implicitly by `Pad.Pad(obj)`, if `obj` defines `userclose()`.

```
    class PadRcv {
public:
virtual void    cycle();                /* see Pad.alarm() */
virtual void    linereq(long,Attrib=0); /* see Pad.lines() */
virtual int     accept(Action);         /* see Pick()      */
};
```

5. Class Pad

Nearly all communication with the terminal is through class `Pad`:

```
class Pad {
        PadRcv   *object;
public:
                Pad(PadRcv *);
                ~Pad();
        void    banner(char* ...);
        void    name(char* ...);
        void    options(Attrib, Attrib=0);
        void    makecurrent();
        void    menu(Index);
        void    menu(Menu&);
};
```

`Pad.Pad(obj)` initializes a pad bound to `obj.` The default attribute `FOLD` is set. The attributes `ACCEPT_KBD` and `USERCLOSE` are set, if `obj` defines `kbd()` and `userclose()`, respectively. Default name and banner are set to `'<name>'` and `'<banner>'`. The name appears in the list of pads at the bottom of the button 3 menu. The banner appears at the top of the pad's window. `Pad.~Pad()` removes all record of the pad; the window in the terminal vanishes. `Pad.name()` and `Pad.banner()` take `printf` arguments to redefine the name and banner. `Pad.options(on, off)` switches on the first set attributes and then switches off the second (optional) set. The attributes for pads are:

```
typedef unsigned short Attrib;

#define FOLD            ((Attrib) 0x.... )
#define TRUNCATE        ((Attrib) 0x.... )
#define SORTED          ((Attrib) 0x.... )
#define ACCEPT_KBD      ((Attrib) 0x.... )
#define USERCLOSE       ((Attrib) 0x.... )
#define DONT_CLOSE      ((Attrib) 0x.... )
```

Each is represented by a separate bit; sets are composed by bitwise or. `FOLD` specifies that lines longer than the width of the window be folded so that the whole text is displayed. `TRUNCATE` specifies that long lines be truncated at the right. These can be overridden as individual lines are inserted into the pad. The user can also override any host setting on pad or line. `SORTED` specifies that the lines in the pad be displayed by dictionary order rather by numeric key. For the effect of `ACCEPT_KBD` and `USERCLOSE` see `PadRcv.kbd()` and `PadRcv.userclose()`, respectively. `DONT_CLOSE` inhibits the user from closing the window; attempts are still reported if `USERCLOSE` is set. Combinations: `DONT_CLOSE|USERCLOSE` means the host has complete control; `USERCLOSE` means the user can close unilaterally and the host is told it happened; `DONT_CLOSE` means the user can't close and the host doesn't care; neither means the user can close and the host isn't told.

`Pad.makecurrent()` forces the terminal to bring the pad's window to the front of the screen and select it as current. If no window exists for the pad – the initial state – the terminal solicits the user (once) to sweep out a rectangle. `Pad.menu()` takes a `Menu` (or a menu's `Index` – ignore `Index` on first reading) and binds the corresponding menu to the pad.

```
class Pad {
public:
        void    insert(long, Attrib, PadRcv*, Index, char* ...);
        void    insert(long, Attrib, PadRcv*, Menu&, char* ...);
        void    insert(long, Attrib, char* ...);
        void    insert(long, char* ...);
        void    removeline(long);
        void    clear();
        void    tabs(short);
        void    lines(long);
        void    createline(long, long);
        void    createline(long);
};
```

`Pad.insert()` puts a line of text into a pad. The first argument is a non-zero 32-bit key. The trailing `printf` arguments are the text, which is truncated at about 250 characters. Only lines corresponding to keys for which strings have been inserted are displayed. Line `1` is displayed at the top of the window and line `2**32-1` at the bottom. The second (and subsequent) insertions for a given key overwrite the original. `Pad.removeline(k)` deletes the line with key `k`. Tab stops are set every 8 characters. The default is overridden for each pad individually by `Pad.tabs()`. Non-printable characters other than tab are replaced by blanks.

The attributes that may be set on a line are:

```
typedef unsigned short Attrib;

#define SELECTLINE      ((Attrib) 0x.... )
#define ACCEPT_KBD      ((Attrib) 0x.... )
#define FOLD            ((Attrib) 0x.... )
#define TRUNCATE        ((Attrib) 0x.... )
#define DONT_CUT        ((Attrib) 0x.... )
#define USERCUT         ((Attrib) 0x.... )
#define FLUSHLINE       ((Attrib) 0x.... )
```

`SELECTLINE` causes the line to be selected as though the user had pointed to it. `FOLD` and `TRUNCATE` override the pad. `DONT_CUT` makes the line immune to the `'cut'` and `'sever'` terminal operations that remove lines; `USERCUT` calls `PadRcv.usercut()` if the line is cut; they behave like `DONT_CLOSE` and `USERCLOSE` on pads. `FLUSHLINE` prevents the line from being buffered in the host, but should be needed very rarely. If `Pads`' buffering heuristic works, the application shouldn't know about this.

An object is bound to a line by the `PadRcv*` and `Menu&` (or `Index`) arguments. This is just like binding to a pad, except that no attributes are set implicitly.

`Pad.lines(n)` changes the pad's key space from `1..2**32-1` to `1..n`. It also makes the terminal partially responsible for deciding what lines should be inserted. The host can still insert lines at will. As the user scrolls the window, a call `linereq(k,0)` instructs the pad's object to insert line `k`. So `linereq()` should be defined:

```
class PadRcv {
public:
virtual void linereq(long, Attrib=0) { /* do nothing */ }
};
```

There can be no assumptions made about the order in which calls to `linereq()` will occur. The terminal caches only a bounded number of lines and (re-)requests them from the host as the user scrolls the window. However, a common pattern of calls is

```
linereq(m,0), linereq(m+1,0) ... linereq(n-1,0), linereq(n,0)
```

for which it may be worth optimizing. The `Attrib` argument is included for the convenience of calls from host code.

In pads for which `lines()` has not been called, `Pad.createline(i)` inserts a dummy line with key `i` and `Pad.createline(lo,hi)` inserts a set of dummy lines with keys `lo..hi`. `linereq()` is called if any of these lines is to be displayed. (Theoretically, `createline()` subsumes `lines()`, but the current implementation is inefficient for large key ranges.)

`Pad.clear()` removes all lines from the pad.

The terminal's real-time clock can be used to invoke host operations periodically.

```
class Pad {
public:
        void    alarm(short=0);
};

class PadRcv {
public:
virtual void    cycle();
};
```

At least n seconds after `pad->alarm(n)` is called, `cycle()` is called on `pad`'s object. For a given pad, if there is an outstanding call to `alarm()`, results are undefined for further calls to `alarm()` before `cycle()` is called. `cycle()` is called for `alarm(0)` as soon as all currently outstanding operations from the user and pending `cycle()`'s have been completed. Thus an arbitrary number of host objects can use `alarm(0)` for round-robin scheduling, with interleaved user operations.

6. Class menu

Class `Menu` is used to build menus. Menus are bound to windows and lines by `Pad.menu()` and `Pad.insert()`.

```
typedef void (*Action) (...);

class Menu {
public:
                Menu();
                ~Menu();
    void    first( char*, Action, long=0 );
    void    last(  char*, Action, long=0 );
    void    sort(  char*, Action, long=0 );
};
```

Objects of type `Menu` are usually declared as automatic (stack) variables. The lifetime of

```
Menu m;
```

need only be long enough to bind it to a window by

```
pad->menu( m );
```

or to a line

```
pad->insert( ... m ... );
```

`Pad.insert()` and `Pad.menu()` extract the current value of `m`; they do not retain a reference to `m`. Subsequent changes to `m` are not reflected in the menu presented for those objects to which it was bound earlier.

The void constructor initializes an empty menu. `Menu.first()`, `Menu.last()` and `Menu.sort()` add entries to the menu. Their `char*` argument is the string to appear as one entry in the menu presented to the user. The string can be temporary; `Menu` takes a copy of it. The `Action` argument is the function to be invoked if the entry is selected. The optional `long` argument is to be passed on to that function as an argument. The arguments

```
( "next year", (Action) &calendar, 1985 )
```

would result in the call

```
object->calendar(1985)
```

which is implemented as the equivalent of the C call

```
calendar(object,1985)
```

The `object` is the `PadRcv*` pointer passed to either `Pad.Pad()` or `Pad.insert()`.

At any time, a `Menu` is a sequence of such entries. `Menu.first()` adds a new entry to the beginning of the sequence. `Menu.last()` adds a new entry to the end. `Menu.sort()` inserts before the first string lexically greater than its argument. The *order* of a set of calls to these functions is important. For example,

```
Menu m1, m2;

m1.first( "z" ... ); /* z */
m1.first( "x" ... ); /* x, z */
m1.last(  "a" ... ); /* x, z, a */
m1.sort(  "q" ... ); /* q, x, z, a */
m1.sort(  "p" ... ); /* p, q, x, z, a */

m2.sort(  "q" ... ); /* q */
m2.sort(  "p" ... ); /* p, q */
m1.first( "x" ... ); /* x, p, q */
m2.first( "z" ... ); /* z, x, p, q */
m2.last(  "a" ... ); /* z, x, p, q, a */
```

A second constructor initializes a menu to have a single entry:

```
class Menu {
public:
        Menu( char*, Action, long=0 );
};
```

If an entry string ends with a question mark, e.g.

```
m.first( "delete all?" ... );
```

the terminal makes the user confirm the selection before passing it to the host.

The leftmost character of the string with bit `0200` set is treated as a ‘‘padding’’ character. The padding character `0200|'c'` is replaced by sufficiently many copies of `'c'` to make the string as wide as the widest string in the menu. For example, the strings

```
"a\240reg"
"table\240arg"
"\255"
"main()\25675"
"f()\256123"
```

would appear in a menu as

```
a     reg
table arg
---------
main().75
f()...123
```

`040`, `055` and `056` are `' '`, `'-'` and `'.'`, respectively. (The expansion is done by `menuhit()` in the terminal .)

7. More about Menus

Skip this section on first reading.

Rebuilding a menu every time it is needed does not create a new copy of the menu in the host or terminal. For each distinct menu entry, the incremental memory used in the terminal is the size of the string. For each distinct menu, the incremental memory in the terminal is 2 bytes per entry. More memory is used for host data structures, but linear in distinct entries and menus. For most applications it should not be necessary to worry about the way menus are implemented. The application programmer’s effort ought to go into composing good menus. A built-in pad can be instantiated by `NewPadSats()` to report how much memory is being consumed by menus.

If profiling indicates that too much time is consumed on the host by repeatedly building the same menu, it is possible to obtain a menu’s *index*, the key by which it is known internally to Pads. Once extracted, this key may be passed repeatedly to `Pad.menu()` and `Pad.insert()`.

```
class Index {
public:
    int null();
};

class Menu {
public:
    Index   index(char* =0, Action=0, long=0);
};

class Pad {
public:
    void    insert(long, Attrib, Univ*, Index, char* ...);
    void    menu(Index);
};
```

The representation of the null `Index` is zeros. A statically allocated `Index` is initially null, tested by `Index.null()`. This permits the following style of code, in which the menu is built once, but used many times.

```
void Any.linereq( long k, Attrib a = 0 )
{
    static Index ix;

    if( ix.null() ){
        Menu m;

        m.first( ... );
        .
        .
        .
        ix = m.index();
    }
    pad->insert( k, a ... ix ... );
}
```

Reference to a pre-computed menu can also be inserted into another menu.

```
class Menu {
public:
    void    first(Index);
    void    last(Index);
};
```

A menu is more than a sequence of atomic entries, as stated above. Each member of the sequence can also be another menu. This recursive structure permits arbitrary trees.

If arguments are given to `Menu.index()` the menu is treated differently when incorporated in another menu. The string appears in the parent menu as a regular entry with an associated action. The sub-menu is only reached if the user explicitly selects it to be displayed.

`NumericRange()` makes it easier to build numeric (sub-)menus.

```
Index NumericRange(short, short);

class PadRcv {
public:
virtual void    numeric(long);
};
```

```
Index ix = NumericRange(lo, hi);
```

is (almost) equivalent to

```
Menu m;
for( long i = lo; i <= hi, ++i ){
        char dec[BIG_ENOUGH];
        sprintf( dec, "%d", i );
        m.last( dec, &numeric, i );
}
Index ix = m.index();
```

Of course, no such menu is built; the numeric range is represented by the pair of shorts. Moreover, the binding of `numeric()` occurs through the regular virtual function calling mechanism when the operation comes back from the terminal, not in the menu construction.

8. Miscellany

```
void NewHelp();
void NewPadStats();
long UniqueKey();
void PadsWarn(char* ...);
```

`NewHelp()` instantiates an object that creates a ‘‘help’’ pad, presenting a synopsis of the Pads manual page.

`NewPadStats()` creates a pad that shows consumption of host and terminal memory.

Each call of `UniqueKey()` returns a distinct value, monotonically increasing (at least to begin with.)

`PadsWarn()` takes `printf` arguments and writes a message to the user. It use is discouraged for routine communication with the user.

9. References

1. B.S. Stroustrup, ‘‘C++ Reference Manual’’, Computing Science Technical Report CSTR-108, AT&T Bell Laboratories, 1984.