THE PETIT-AMI TOOL KIT

VERSION 0.1

Contents

1 Introduction: What is Petit-Ami

Petit-Ami is a general purpose tool kit designed for C, but enabled with bindings for several languages and thus not restricted to a particular language. Petit-Ami was the toolkit for the language Pascaline, an advanced Pascal compatible language. I ported it to C for the following reasons:

1. This makes it easier to adapt to other languages outside Pascaline.
2. It is easier to integrate with today's operating systems and general code base, which is for the most part written in C.
3. Programmers will be more likely to use and adapt a system written in C, which has universal acceptance.

Petit-Ami is unlike many graphical TKs in use today in that:

1. It does not force you to "change models" from standard line oriented or even terminal oriented code. Instead of introducing a new, windowed graphical paradigm, Petit-Ami works with the code you have, immediately adapts it to any target environment, and then lets you add terminal, or graphical, or advanced widget controls at will, complementing the existing code.
2. Petit-Ami does not force you to use or emulate an object oriented code model. Although it is common to add an object model layer atop Petit-Ami, it is not required.
3. Petit-Ami does not force you to use callbacks. Callbacks can break the linear structure of the code, and they are optional in Petit-Ami. Further, several languages allow for nested functions, which makes using callbacks difficult.
4. Petit-Ami is designed to be language agnostic. It does not use void pointers, rely on casting to represent multiple types, etc. Only the constructs that make sense for all languages are represented.
5. You can use any part of Petit-Ami you wish. Each module of Petit-Ami is separate and does not rely on the others. You can use just one module, or some of the modules, or all of it. It's your choice.

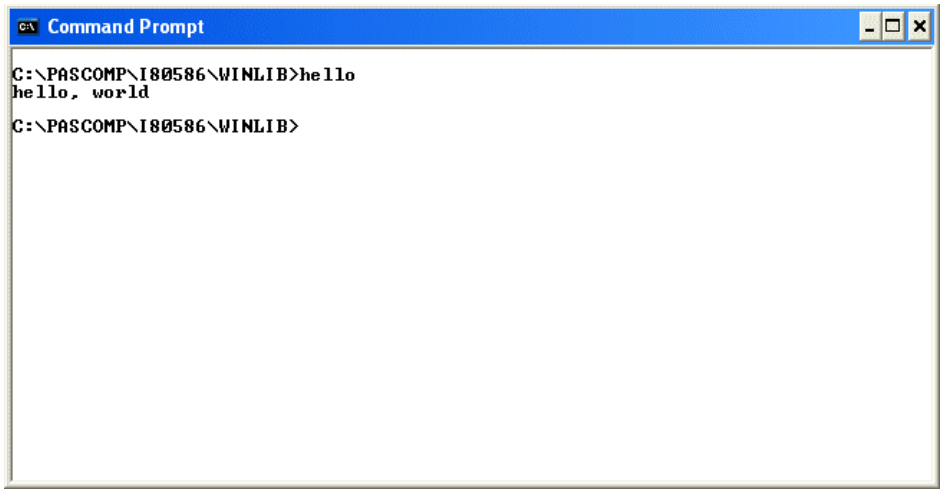
The first and most visible aspect of Petit-Ami is that any C program, graphical or not, can be immediately compiled and run with it. The only dependency is the C standard ANSI library for it, commonly called the "whitebook" standard.

The standard "hello, world" program is used as an example. This program, like any other program written in C that does not already include graphics or terminal controls, can be redirected to any supported operating system, and any model of line oriented, terminal oriented, full graphical display, and windowed environment of any of text mode or graphics mode. If the same CPU is targeted, only a relink is required, not a full recompile.

The hello world program:

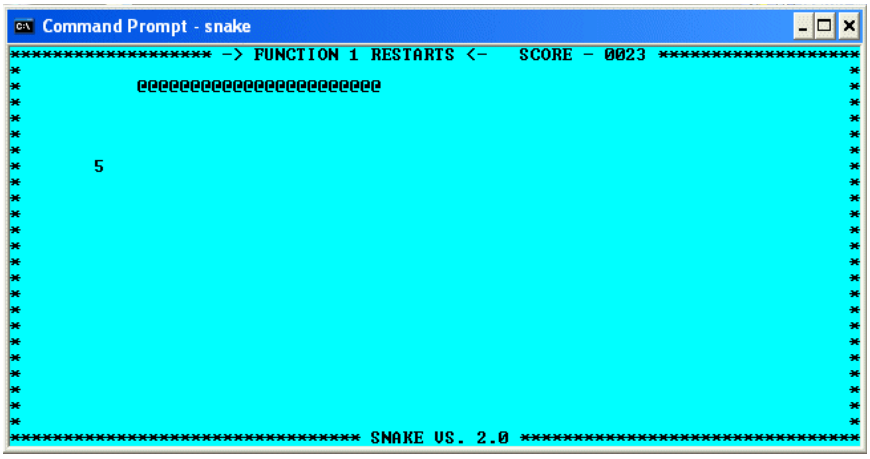
```
#include <stdio.h>

program hello(output);

begin

   writeln('hello, world')

end.
```

Hello world for Windows command line.

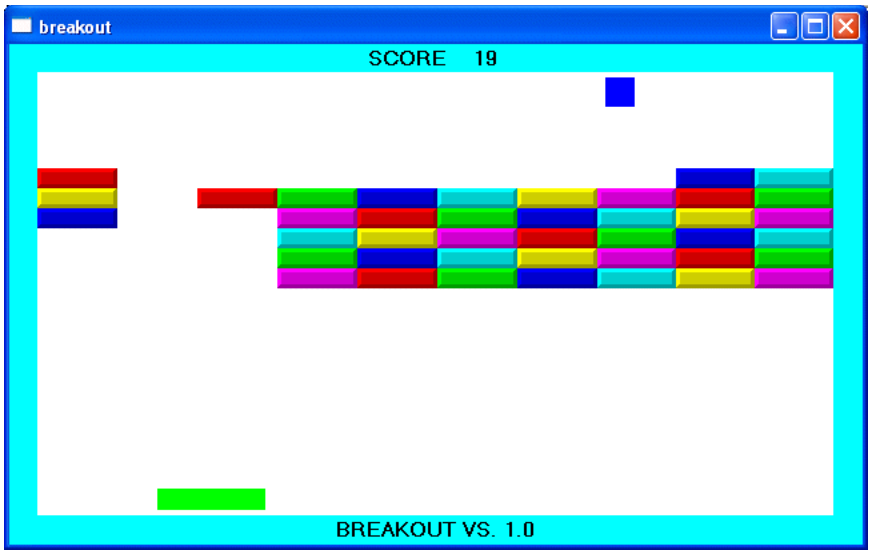


Hello world for Windows console API.

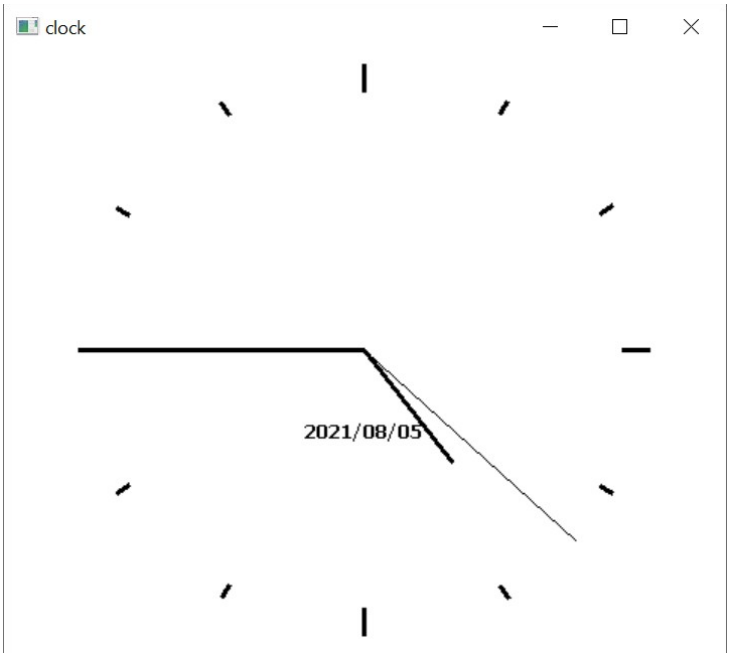
Hello, world compiled for graphical mode.



Console game.



Breakout graphical game with sound and joystick.



Clock program, resizable.

Petit-Ami is different from other TKs in that it tries to provide a complete runtime environment for the programs it ports, vs., just providing graphics. This is because it is designed as a the working library for a language installation (Pascaline), as well as the fact that I don't believe you should have make a collection of unrelated modules just to get programs to run.

1.1 History

The roots of Petit-Ami go back to 1982, when I worked on a Unix system for the first time (Unisoft, using 68000 CPUs). I wanted to create a library with a wrapper around directory listing that would be universal, so that I could write programs for any operating system and just change out the library. That system grew to cover a range of operating system functions, and that became the **services** module here in Petit-Ami. I moved that to MS-DOS on an IBM-PC, and used it there for many years.

In about 1984, I acquired a Wyse 80 terminal, a large format terminal capable of presenting about 120 columns of text. I didn't want to directly program for it, and so created the second module in the Petit-Ami series, which became the **terminal** library in Petit-Ami.

In 1995, I decided to move my programming work to Windows 95, mainly because it was the first Windows version to support a 32 bit API. **terminal** was then ported to the Windows console mode API. In 1996, I wanted to follow that up with a full graphical API using the Win32 API as a base.

At that time, there were several Graphical Tool Kits coming out. Most of them discarded any backward compatibility and adopted an object oriented model for the display, and used liberal amounts of callbacks. This was a big factor because the Win32 API made it very difficult to program without callbacks. I didn't see any reason to completely dispose of backward compatibility, nor to rely heavily on callback models, or to force use of object oriented models. I don't have anything against object oriented programming, and use it frequently. I just don't believe programming is improved by forcing people to use new models. It should be optional, just as C++ is (mostly) upward compatible with C.

Thus I started with a win32 API version of terminal, which was very difficult at that time, since Windows was built extensively around the callback model. For my guide, I used the Xwindows model of an event loop as the basis. The result was a fully graphical TK which was fully upward compatible with the terminal model. I added on to the graphics interface, and later I added modules for sound and networking, and that fills out the Petit-Ami set of libraries.

In 1996, most people making TKs were working with the idea that text based interfaces were going to be obsolete very soon. Everything would be point and click. Graphical interfaces would discard any backward compatibility, since text mode would be useless very soon. There were a couple of libraries that did feature upward compatibility for text mode programs, but all they did is emulate text mode programs in Windows, with virtually no path to go beyond that.

In the years since, there has been a renaissance for text processing, as programmers realized that command line interfaces (CLIs) were more powerful and more concise in the majority of cases. Indeed, Linux features command line operations for virtually everything that it's graphical applications can do, and when a feature comes out in Windows, usually the first thing programmers request is a command line version of the same functionality. It's not just about the power of CLIs, but also the fact that CLIs are scriptable, and have the ability to be included as components into larger programs without change.

It is for these reasons that I think Petit-Ami fits well with modern architectures. It's not so much that I think a lot of text mode programs will be converted to full graphical using Petit-Ami (although that is certainly possible), but more that Petit-Ami makes it effortless to cross from text mode programs to graphical mode programs (and back).

At the same time, Petit-Ami is a good graphical model as well. It is dramatically simpler than direct programming to the common operating system interfaces such as Win32 and Xwindows (as indeed most TKs are), and it is built on a series of timeless paradigms that fit together well.

2 Reliance on ANSI C/libc

The base Petit-Ami code relys on both ANSI C as well as the standard C library, generally referred to as libc. This means that systems that support both of these standards will be able to host Petit-Ami.

3 Coining

To prevent namespace collisions, each of the calls in Petit-ami are coined with the prefix “pa_” such as:

```
pa_list();
```

Language bindings that allow for separate namespaces (such as Pascaline or C++) usually drop the pa_ prefix.

4 Module format

Each module of Petit-Ami has both a .c file and a .h file, each of which has an include guard (prevents multiple inclusions of the same file). Each module has a startup section and a ending section that starts before the program starts, and ends after the program ends, respectively. It is used to set up global variables used in each module.

All of this, of course, is simply good programming practices, and it is covered in the document “C Language Coding Standard”, which is included with the Petit-Ami release.

5 Overview of Petit-Ami libraries and modularity

The modules in Petit-Ami are divided into two basic types:

1. Basic extensions used by programs regardless of extra devices or capabilities available.
2. Extensions that introduce new device capabilities.

The following extension libraries will be shown in Petit-Ami:

Section 7: Service library - Directory lists, file name handling, paths, environment, program execution, date and time, internationalization.

Section 8: Terminal library - Output to text surface, advanced input.

Section 9: Graphical library - Output to graphical surface.

Section 10: Windowing management library - Management of multiple windows.

Section 11: Widget library - Buttons, lists, dialogs relevant to screen based programs.

Section 12: Sound library - Midi and wave input and output.

Section 13: Network library - Network program access.

Typically the windowing library and the widget library are combined with lower level modules. For example, the graphical library, the windowing library and the widget library are combined on systems that are capable of all three functions.

5.1 Services

services.c/.h contains a series of operating system support extensions such as filename/path handling, directory listings, time and date, environment strings, executing other programs, and similar functions.

5.2 Advanced User I/O and Presentation Management

The advanced user I/O modules are a family that implement a series of advancing levels that match advancing levels in I/O capabilities, including:

Terminal character presentation

Gives the ability to place characters on a grid for a typical fixed size character display.

Graphical presentation

Gives the ability to draw figures, and place characters with proportional fonts.

Window management

Divides the display surface into a series of independent windows.

Widget placement and management

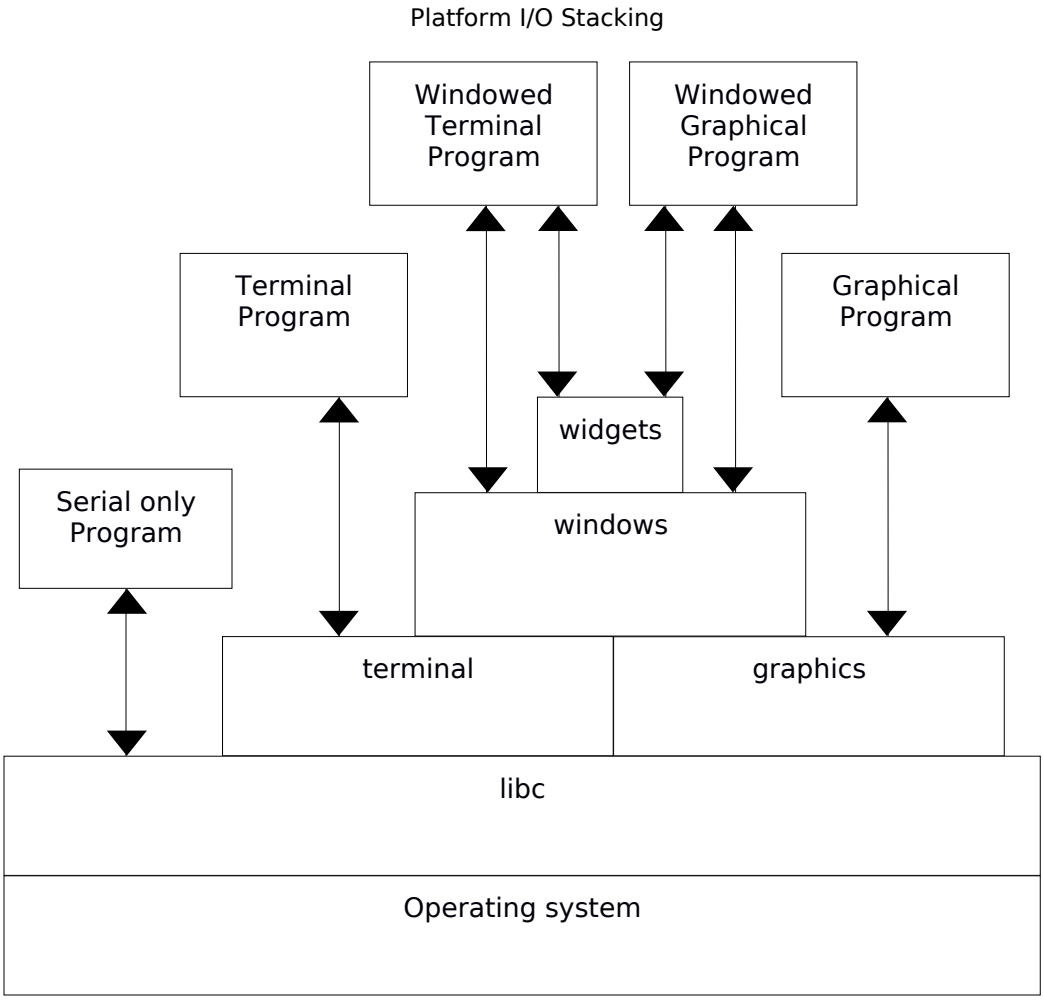
Gives a library of common controls, layered components and dialogs.

All of the supported display modes are upward compatible, and fixed size character presentation is always available in any graphic mode. The result is that there is two possible stacking levels of presentation:

```
Terminal  -> Single fixed display -> Windowed display -> Widgets
Graphical -> Single fixed display -> Windowed display -> Widgets
```

The terminal, graphical, window management and widget libraries are a set of libraries that implement a series of screen surface operating standards in an ascending sequence of capability:

These layers are all forward and backward compatible, so that any lower level can be used in any higher level. For example, a ANSI C program that only inputs and outputs in terms of characters formatted in terms of lines can be run under any level all the way up to a multiple windowing environment with widgets without source change.



The primary method for performing this is the use of the "character grid", which shows where the cells of screen characters exist in a graphical system. The character grid is only relevant to characters as a figure, and not to lines or other figures, and output to the grid is the default. A program use arbitrary character placement by explicitly specifying it, and it can turn off the automatic character wrapping features of the grid.

Similarly, all text and graphical windows are buffered in a multiple windowing system, so that they need not be aware of the windows management unless they wish to be.

Serial I/O is the name given here of the default method of ANSI C which goes back to its origin, and includes the **printf()**, **fscanf()**, and similar library procedures.

Terminal I/O is so named after the terminals that were used with computers for several decades, and for text mode still in use even in windowed operating systems.

From here, two distinct branches exist, one with, and one without graphics capability. For example, it is possible to have both multiple windows, widgets and dialogs all while staying within the capability of a character I/O only terminal. Therefore, there exists both a graphics version of the windows management library and the windows library, and one of each with character only ability.

The names of the libraries, as appears in a uses or joins statement, is as follows:

Name	Contents	Call model
terminal	Terminal I/O	Terminal
graphics	Graphical I/O	Graphical
windows	Managed windowing	Terminal and graphical
widgets	Widgets and dialogs	Terminal and graphical

The serial level does not require an explicit library, since that is the normal I/O method specified in ANSI C.

The key to understanding the variation in libraries is that a use of a particular library does not necessarily lead to a completely different block of code. A system that only has graphical output modes will implement the terminal library by aliasing that to the graphical library. It is also common for a graphical or terminal mode library to include all of the windows management and widget/dialog functions in the same module. Thus, a use of the windows management libraries does not cause a specific library to linked as much as flag that the program will be using these features.

5.3 Naming

The functionality of the I/O modules nest, which effectively means that each of the I/O modules extend each other. Unfortunately, there is no mechanism in ANSI C for modules to extend one another. The net effect is that when a module is referenced that is a base class of a static module, that reference is aliased to the derived class. For example, references to calls in **terminal** are rerouted to **graphics**, since graphics contains all of the procedures and functions of terminal.

These combinations of Petit-Ami modules are possible:

Module	Extended to
terminal	**windows**
terminal	**widgets**
graphics	**widgets**

The method this is carried out is implementation specific. Each of the I/O modules has its own interface specification, which includes a superset of the base module. This allows each module to be fully checked against its own specification. A program that specifies **terminal** only sees **terminal** definitions, a program that specifies graphics only sees **graphics** definitions, and so forth. At the lowest level (assembly language or “linker” level), aliases are provided for the module names, and so a program compiled to the **terminal** standard can be linked to a graphics module, etc.

5.4 Advanced device libraries

5.4.1 sound

sound gives the ability to drive input and output MIDI messages, perform sequencing, and input and output wave files.

5.4.2 network

network allows access to a network using the ANSI C library file model. It supports TCP/IP, SSL, and messaging.

5.5 Library procedure and function notation

The libraries description continues the idea that program examples be compilable from the source material. The parts of the interface definitions have been broken into sections by the same name. For example, terminal contains several source modules named terminal, and the understanding is that these are different sections of the same interface module.

6 Using C++ With Petit-Ami

C++ can be used with PA and is downward compatible. Everything that works with C works with C++ as well, with the exception of stream I/O. Some systems can use stream I/O and some cannot. See the details on your specific operating system[1].

6.1 Two Modes for C++ Programs

C++ can be used in two different levels, which I refer to as C+ and C++ levels. These levels use the file endings .cp and .cpp, respectively[2]. The C+ level uses namespaces and overloads for functions. The C++ level implements this plus object orientation.

The reason for the two levels is that many programmers use C++ as an “extended C”, which is to say, they use procedural oriented code supplemented by the features of the C++ language. For PA, this means the namespace feature of C++, and the ability to overload functions.

The reason for two different file endings, .cp and .cpp is that there are examples in the PA tree that use both the C+ mode and the C++ mode in the same program. So having the different file endings makes it clear what mode is being used in the source file.

6.2 C++ Extention layer

The C+ and C++ support is implemented as a wrapper in C++ to the C libraries in PA. The code for the wrappers is in the cpp directory. The header files used are in the hpp directory. To include a particular library in C++, you use the hpp version of the header file:

```
#include <stdio.h>
#include <stdlib.h>
#include <terminal.hpp>

using namespace terminal;

int main()
{

    evtrec er;
    printf("hello c++ world\n");
    do { event(stdin, &er);
        if (er.etype == etterm) exit(1);
    } while (er.etype != etenter);

}
```

In this case, the terminal.hpp file is the analog of the terminal.h file used in C. In C++, you only use the hpp version of the file.

[1] At this writing, Petit-Ami/Linux works with stream I/O, but Petit-Ami/Windows does not.

[2] You don’t have to use this convention. It is used to separate different Petit-Ami example programs that do the same thing, but use different language modes.

To implement the functionality a fairly thin layer of C++ code is used to form the C++ interface from the underlying C libraries. This code is in the cpp directory.

Each header file contains a namespace corresponding to the library name. In the example for terminal.hpp, it contains a namespace **terminal**. In the client code using it, you either open up the namespace using:

```
using namespace terminal;
```

Or you use C++ namespace notation to access the declarations in the file, like:

```
#include <stdio.h>
#include <stdlib.h>
#include <terminal.hpp>

int main()
{

    terminal::evtrec er;
    printf("hello c++ world\n");
    do { terminal::event(stdin, &er);
        if (er.etype == terminal::etterm) exit(1);
    } while (er.etype != terminal::etenter);

}
```

Or any mix of the above.

6.3 Library documentation for C++

Because C++ can use all of the functionality of C for PA, C++ features are documented as part of the C level documentation for libraries. It is clearly marked what the extended functionality for C++ is.

7 System Services Library

services.c/.h contains common operating system related tasks, including directory access, time and date, files and paths, file attributes, environment strings, the local option character, execution of external programs, location, and internationalization.

7.1 Filenames and Paths

A file specification is composed of a path, name and extension:

<path><name><ext>

The exact format of a file specification changes with the operating system. The routine **pa_brknam()** takes a file specification and breaks it down into its path, name and extension components. The routine **pa_maknam()** does the opposite, creating a composite name from the three components. When a file specification has no path, it means that it refers to the default path. When a file specification needs to be printed or saved in an absolute format, with the path always specified, the **pa_fulnam()** routine is used to "normalize" the file specification by filling out the complete path.

To parse file specifications from the user, the routine **pa_filchar()** returns a character set of all possible characters in a filename. This can be used to get a sequence of characters from a string or file that can constitute a file specification. Once the potential file specification has been loaded into a string, it can be checked for validity as a file specification by **pa_validfile()**, and as a path by **pa_validpath()**.

Filenames are based on the idea that there is a set of characters that may be used for filenames in a particular installation, and the characters outside that set are used to delimit between filenames. This characteristic of filenames allows a program to load the filename into a string, then check it’s proper structure using subsequent calls. The program may remove certain characters from the set of filename characters to be able to parse command lines.

A common method used to represent filenames in command lines and other text when the characters allowed in a filename are essentially unlimited is to quote the filename. Using this method, the filename appears as:

“myfile”

or

‘myfile’

This can work together with limited character set filenames by recognizing the leading quote. If both types of quotes are allowed, then the leading quote should match the trailing quote. Additionally, the program should be able to recognize a “quote image” of the forms:

Character sequence	Result
“”	Double quote “
‘’	Single quote ‘
\”	Forced double quote “
\’	Force single quote ‘

A robust program would recognize all of these forms.

Note that **services** does not contain any method to parse filenames.

If a file specification contains wildcards, this can be determined by **pa_wild()**. **services** does not define what wildcards characters or specifiers are used, or their format. **pa_wild()** simply indicates that the filename contains a wildcard specification that may result in multiple files being indicated by the same file specification.

7.2 Predefined paths

To find common objects that a program needs, three predefined paths are provided:

Program path

> Is the path that the program was executed from. This is used to find data that accompanied the program, and system wide option files. **pa_getpgm()** is used to read the path.

User Path

> This is the path for the current user's home directory. This is used to store options that only apply to the current user. **pa_getusr()** is used to read the path.

Current Path

> The default path is used to find options that apply only to the current file being worked on. Unlike the other path types, the current path can be both read and set. Setting the current path will set the default path used to finish incomplete file specifications. **pa_getcur()** is used to read the path, and **pa_setcur()** is used to set it.
>
> Note that if the system has no concept of a current path, this property is used to form full path names from default path names in **services**.

7.3 Time and Date

Time is kept in two different formats in **services**. The first is seconds time, and the second is "clock" time. Seconds time is literally a count of the number of seconds since a fixed reference time. Clock time is a free running clock that ticks every 100 Microseconds.

Seconds time is returned by the **pa_time()** function. It is in a format called "S2000", and it is the number of seconds relative to midnight on the morning of January 1, year 2000. This means that S2000 has a negative value for years before 2000, and a positive value after that. S2000 time is dependent on the representation of a **long**, which follows the size of the word in the current CPU. This results in the following limits according to bit size:

Bits	Farthest year into the past	Farthest year into the future
32	1932 AD	2068 AD
64	292471206678 BC	292471210678 AD

This represents a reasonable range of time for 32 bits, and by the year 2068 it is likely that the time will universally be 64 bits or better. Note that it does not matter how many bits are returned by time, so that transition will occur seamlessly. Because 64 bits is already in excess of what is needed for computer

based timekeeping, it is likely that time in 64 or more bits will actually contain fewer bits, despite the **long** representation.

S2000 time is self-relative, meaning that times relative to the year 2000 are measured by a fixed number of seconds. S2000 only matched UTC time once, exactly at midnight on the morning of January 1, year 2000. All times after that or before that or after that are increasingly diverge from UTC. In particular, this means that the current time and date will not match current UTC.

There are two types of calculations that can be applied to UTC, static and dynamic. The static calculation typically finds UTC by applying leap year corrections, then a fixed table of leap second years. The static calculation will fall out of accuracy from the time that the system is released. It is impossible for it to be otherwise, since UTC is based on astronomical observation.

The dynamic calculation relies on receiving a current table of corrections from the network or other communications method. This is the same calculation as fixed time, but with a continually updated table of leap seconds.

When an S2000 time is not available, it is customary to set it to –**LONG_MAX**, which makes it clear that the time was not set.

The time returned by **pa_time()** is in GMT or "universal" time. To convert to local time, the function **pa_local()** is used. It takes the given GMT S2000 time, and offsets it by the local time zone offset, and by daylight savings time, and returns the adjusted local time.

The time can be placed, in character format, to a string by **pa_times()**, and written to either an output file, or the **stdout** file by **pa_writetime()**. The date can be placed, in character format, to a string by **pa_dates()**, and written to either an output file, or the standard output by **pa_writedate()**. These procedures format the time and date using the current internationalization settings of the host.

pa_clock() time is typically derived from a free running counter kept in the host computer, and it is represented by **long** values. It may or may not be synchronized to the values returned by time. For this reason, **pa_clock()** should be treated as self-relative and not compared to **pa_time()** in any way.

pa_clock() time is treated differently from **pa_time()**. Since it free runs, you must be prepared for it to "wrap", or suddenly start counting up from zero. This can be determined by if any stored time is greater, or later in time, than the current clock value. The function **pa_elapsed()** takes a reference time, and determines how much time has passed from that, including compensation for wraparound. The exact count at which the **pa_clock()** count wraps is system dependent.

The total amount of time that can be represented with **pa_clock()** is determined not only by the bit size of the **pa_clock()** return value, but also by the size of the counter the host computer maintains. All that is guaranteed is that clock will be able to keep a unique time for at least 24 hours.

The actual increment of time for each tick of the clock is determined by the host computer. If the host cannot time to 100 microsecond accuracy, then the **pa_clock()** time will increment in multiples > 1, or effectively a running approximation of the actual timer. For this reason, there may not be an exact time length between successive counts of the timer.

7.4 Directory Structures

The **pa_list()** procedure takes a file specification, including wildcards, and returns a linked list of all of the matching directory entries:

```
/* attributes */
typedef enum {

    pa_atexec, /* is an executable file type */
    pa_atarc,  /* has been archived since last modification */
    pa_atsys,  /* is a system special file */
    pa_atdir,  /* is a directory special file */
    pa_atloop  /* contains heriarchy loop */

} pa_attribute;
typedef int pa_attrset; /* attributes in a set */

/* permissions */
typedef enum {

    pa_pmread,  /* may be read */
    pa_pmwrite, /* may be written */
    pa_pmexec,  /* may be executed */
    pa_pmdel,   /* may be deleted */
    pa_pmvis,   /* may be seen in directory listings */
    pa_pmcopy,  /* may be copied */
    pa_pmren    /* may be renamed/moved */

} pa_permission;
typedef int pa_permset; /* permissions in a set */

/* standard directory format */
typedef struct pa_filrec {

    char*             name;    /* name of file (zero terminated) */
    long long         size;    /* size of file */
    long long         alloc;   /* allocation of file */
    pa_attrset        attr;    /* attributes */
    long              create;  /* time of creation */
    long              modify;  /* time of last modification */
    long              access;  /* time of last access */
    long              backup;  /* time of last backup */
    pa_permset        user;    /* user permissions */
    pa_permset        group;   /* group permissions */
    pa_permset        other;   /* other permissions */
    struct pa_filrec* next;    /* next entry in list */

} pa_filrec;
typedef pa_filrec* pa_filptr; /* pointer to file records */
```

Each directory file entry has the name of the file, along with a series of descriptive data for the file. These are divided into attributes and permissions. An attribute is a characteristic of the file, and generally does not change. Permissions indicate what can be done with the file, and are divided into the user, group and other permissions.

Not all attributes nor all permissions are available on every operating system. If a particular permission or attribute is not implemented on a given operating system, then setting it will have no effect, and reading it will always return unset.

The size of the file is its size in bytes. The allocation is the total space it occupies on the storage medium, which may be different from its size for several reasons. The blocking may be such that the size is rounded up to the nearest block. The operating system may have the ability to reserve space for the file beyond what it is currently using, or may not release space back to the free space pool if the file is truncated.

The times of interesting events in the file's life are available, in "S2000" format (already discussed). If a particular time is not available, then it is set to –**LONG_MAX**.

The file structure is a collection of items that may be implemented on any given operating system. The way to prevent the need to decide what is and what is not implemented on a particular system is to focus on what is essential for all systems. For example, the size of a file is usually present, as well as the last modification time. The last modification time can be used to determine when to back up files, by comparing it to the date of the backup copy of the same file, or to the modification date of the archive containing the file. Similarly, a "make" style program can determine when to remake a file by looking at the modification time.

7.5 File Attributes and Permissions

The attributes of a file can be set by name with the **pa_setatr()** command. It takes a set of attributes and the filename, and sets all the given attributes on the file. The routine **pa_resatr()** resets attributes. The user permissions for a file are set and reset by **pa_setuper()** and **pa_resuper()**. The group permissions are set and reset by **pa_setgper()** and **pa_resgper()**. The other permissions for a file are set and reset by **pa_setoper()** and **pa_resoper()**.

7.6 Environment Strings

The environment is a collection of strings that is kept by the executive, and passed to programs when they are started. Each string has a name and a value, both of which are arbitrary strings. An environment string can be retrieved by name by **pa_getenv()**, and set by **pa_setenv()**. The entire environment string set can be retrieved at one time by the **pa_allenv()** routine, which uses a linked list to represent the environment strings:

```
/* environment strings */
typedef struct pa_envrec {

    char* name;    /* name of string (zero terminated) */
    char* data;    /* data in string (zero terminated) */
    struct pa_envrec *next; /* next entry in list */

} pa_envrec;
typedef pa_envrec* pa_envptr; /* pointer to environment record */
```

Both the characters and format of the name of an environmental variable and the data it contains are operating system dependent. However, a subset of naming conventions will work on the vast majority of systems. It starts with a character in the sequence 'A'..'Z', 'a'..'z', '_', followed by any number of characters in the sequence 'A'..'Z', 'a'..'z', '_', '0'..'9'. The data in the string can be a series of any valid characters.

The reason for retrieving the entire environment is to pass it on to other programs in an **pa_exec()** statement.

Although most available operating systems implement the environment string concept, it has fallen out of favor in modern programs. Placing the needed configuration strings for a particular program into a place where the executive will use it both requires special calls, and places individual program data into a pool where it can be deleted or corrupted.

Some current systems use a repository concept where program data is kept in a central tree structured database. This is not covered in **services**, and would be covered in another library.

A better method is to use a file containing the configuration information for the program in its startup directory, then optionally another version in the user directory, and finally one in the current directory. This allows options that affect all runs on the current machine to be kept in one file, while the options for a particular user are kept in another file, and lastly the options in use in the current project in the current directory. This system has the advantage that it addresses concerns in a multiuser operating system, and allows each program to maintain its own startup data.

For systems that do not implement an environment string capability, Petit-Ami will commonly keep a file in the user path area that contains the strings. This is then used just for that program, that is, the set of environment strings are kept just for the accessing program.

7.7 Creating or Removing Paths

A file path, or directory, is created by **pa_makpth()**, and removed by **pa_rempth()**. If the directory has files in it, then it cannot be removed until all the files (and directories) under it have been removed. This means by implication that each section of a path must be removed separately, and a tree structured delete would have to repeatedly remove the contents of one element of the path, then remove the path section itself, and so on.

7.8 Option Character

When parsing commands, the option character for the current operating system is found with **pa_optchr()**. **services** does not define the exact format of command line options. The option character is an aid to portability.

7.9 Path Character

The path character is used to separate path components, usually directories in a tree structured file system, within a path for the given system. It can be found with the **pa_pthchr** function. **services** does not define the contents, structure or meaning of a path. The path division character is an aid to portability.

7.10 Location

The functions **pa_latitude()** and **pa_longitude()** exist to give the location of the host computer in geographic coordinates, and return integers. The measurements are ratioed to **INT_MAX**.

The **pa_longitude()** is 0 at the Prime Meridian, a line passing thought Greenwich, UK. The longitudes 0 to **INT_MAX** are east of the line (or in the future, timewise), and longitudes 0 to –**INT_MAX** are west of the line. Thus **INT_MAX** and –**INT_MAX** meet on the opposite side of the world from Greenwich. This means for a 32 bit integer that there is 0.0000000838190317 degrees for each step or 9.3306920025 millimeters per step.

The **pa_latitude()** is 0 at the equator and **INT_MAX** at the north pole, and –**INT_MAX** at the south pole.

The **pa_longitude()** and **pa_latitude()** in minutes and seconds can be found with:

```
void find_dms(int longitude, int latitude,
              int* degrees_longitude, int* minutes_longitude,
              int* second_longitude,
              int* degrees_latitude, int* minutes_latitude,
              int* seconds_latitude,
              int* west, int* north)
{


   *degrees_longitude = longitude*0.0000000838190317;
   *minutes_longitude = longitude%11930465/198841.078518519;
   *seconds_longitude = longitude%198841/3314,0179753086400000;
   if (degrees < 0) *west = 1; else *west = 0;

   *degrees_latitude = longitude*0.0000000838190317;
   *minutes_latitude = longitude%11930465/198841.078518519;
   *seconds_latitude = longitude%198841/3314,0179753086400000;
   if (degrees_latitude < 0) *north = 0; else *north = 1;

}
```

The shape of the world is approximated as an ellipsoid. This means that the circumference of the earth at the equator is a longer distance than the circumference of the earth on the prime meridian (by about 68 kilometers).

The altitude of the host in MSL or Mean Sea Level is given by **pa_altitude()**. It is 0 for the mean surface of the ocean (sea level averaged over a long period of time), and **INT_MAX** at 100 kilometers in height (the altitude at which space begins). It is –**INT_MAX** 100 kilometers in depth.

The altitude of 0 (MSL) is typically defined as height above a model of the geoid, or idealized model of the earth. This means it would have to be calculated against the latitude and longitude to find the actual distance from true center of the earth. However, in the majority of cases it can be accepted as a relative measurement.

The location of the host can be entered by the user or determined automatically by instrumentation (GPS). The host could even be mobile, in which case it is possible to determine speed and direction from the coordinates against time.

If there is no location, it is indicated by longitude equal to –**INT_MAX**. This value of **pa_longitude()** is redundant to **INT_MAX**. Both **pa_longitude()** and **pa_latitude()** are considered unavailable by the value of **pa_longitude()**.

pa_altitude() is available separate from **pa_longitude()** and **pa_latitude()**. It is not available when **pa_altitude()** is –**INT_MAX**.

The country of location is found with **countrys()**, which gives a string corresponding to the English name of the country. The countries of the world, and their associated codes, are given by ISO 3166-1. At this writing, the following countries exist, in alphabetical order:

#	Country	#	Country	#	Country	#	Country
004	Afghanistan	214	Dominican Rep.	430	Liberia	663	st. Martin
248	Aland isl	218	Ecuador	434	Libya	666	st. Pierre
008	Albania	818	Egypt	438	Liechtenstein	670	st. Vincent
012	Algeria	222	El Salvador	440	Lithuania	882	Samoa
016	amer. Samoa	226	Equatorial Guinea	442	Luxembourg	674	San Marino
020	Andorra	232	Eritrea	446	Macao	678	Sao Tome
024	Angola	233	Estonia	450	Madagascar	682	Saudi Arabia
660	Anguilla	748	Eswatini	454	Malawi	686	Senegal
010	Antarctica	231	Ethiopia	458	Malaysia	688	Serbia
028	Antigua	238	Falkland isl	462	Maldives	690	Seychelles
032	Argentina	234	Faroe isl	466	Mali	694	Sierra Leone
051	Armenia	242	Fiji	470	Malta	702	Singapore
533	Aruba	246	Finland	584	Marshall isl	534	Sint Maarten
036	Australia	250	France	474	Martinique	703	Slovakia
040	Austria	254	French Guiana	478	Mauritania	705	Slovenia
031	Azerbaijan	258	French Poly-nesia	480	Mauritius	090	Solomon isl
044	Bahamas	260	French S. Terr.	175	Mayotte	706	Somalia
048	Bahrain	266	Gabon	484	Mexico	710	S. Africa
050	Bangladesh	270	Gambia	583	Micronesia	239	S. Georgia
052	Barbados	268	Georgia	498	Moldova	728	S. Sudan
112	Belarus	276	Germany	492	Monaco	724	Spain
056	Belgium	288	Ghana	496	Mongolia	144	Sri Lanka
084	Belize	292	Gibraltar	499	Montenegro	729	Sudan
204	Benin	300	Greece	500	Montserrat	740	Suriname
060	Bermuda	304	Greenland	504	Morocco	744	Svalbard
064	Bhutan	308	Grenada	508	Mozambique	752	Sweden
068	Bolivia	312	Guadeloupe	104	Myanmar	756	Switzerland
535	Bonaire	316	Guam	516	Namibia	760	Syrian Arab Rep.
070	Bosnia	320	Guatemala	520	Nauru	158	Taiwan
072	Botswana	831	Guernsey	524	Nepal	762	Tajikistan
074	Bouvet	324	Guinea	528	Netherlands	834	Tanzania
076	Brazil	624	Guinea-Bis-sau	540	New Caledonia	764	Thailand
086	Brit. Ind. Oc. Terr.	328	Guyana	554	New Zealand	626	Timor-Leste
096	Brunei	332	Haiti	558	Nicaragua	768	Togo
100	Bulgaria	334	Heard isl	562	Niger	772	Tokelau
854	Burkina Faso	336	Holy See	566	Nigeria	776	Tonga
108	Burundi	340	Honduras	570	Niue	780	Trinidad
132	Cabo Verde	344	Hong Kong	574	Norfolk isl	788	Tunisia
116	Cambodia	348	Hungary	807	N. Macedonia	792	Turkey
120	Cameroon	352	Iceland	580	N. Mariana isl	795	Turkmenistan
124	Canada	356	India	578	Norway	796	Turks isl
136	Cayman isl	360	Indonesia	512	Oman	798	Tuvalu
140	Cent. African Rep.	364	Iran	586	Pakistan	800	Uganda

#	Country	#	Country	#	Country	#	Country
148	Chad	368	Iraq	585	Palau	804	Ukraine
152	Chile	372	Ireland	275	Palestine	784	U. Arab Emirates
156	China	833	Isl of Man	591	Panama	826	U. Kingdom
162	Christmas isl	376	Israel	598	Papua New Guinea	840	U. States of America
166	Cocos isl	380	Italy	600	Paraguay	581	U. States Minor isl
170	Colombia	388	Jamaica	604	Peru	858	Uruguay
174	Comoros	392	Japan	608	Philippines	860	Uzbekistan
178	Congo	832	Jersey	612	Pitcairn	548	Vanuatu
180	Congo, DR of the	400	Jordan	616	Poland	862	Venezuela
184	Cook isl	398	Kazakhstan	620	Portugal	704	Viet Nam
188	Costa Rica	404	Kenya	630	Puerto Rico	092	Virgin isl (British)
384	Cote d'ivoire	296	Kiribati	634	Qatar	850	Virgin isl (U.S.)
191	Croatia	408	Korea DPR	638	Reunion	876	Wallis and Futuna
192	Cuba	410	Korea, Rep. of	642	Romania	732	Western Sahara
531	Curacao	414	Kuwait	643	Russian Federation	887	Yemen
196	Cyprus	417	Kyrgyzstan	646	Rwanda	894	Zambia
203	Czechia	418	Lao DR	652	St. Barthelemy	716	Zimbabwe
208	Denmark	428	Latvia	654	St. Helena		
262	Djibouti	422	Lebanon	659	St. Kitts		
212	Dominica	426	Lesotho	662	St. Lucia		

If the country is not set, an empty string is returned.

The ordinal number of the country is given by **pa_country()**, which returns the number of the language from the table above. These ordinal numbers are given by the standard ISO 3166-1.

The current time zone, is given by **pa_timezone()**. It gives hours in the range of -12 to +14, which indicate the offset in hours from GMT or Greenwich Mean Time. The function **pa_daysave()** is true if daylight savings is in effect in the current host location.

If 24 hour time is used in the current host location, the function **pa_time24hour** will return 1, otherwise 0. The accepted format for 24 hour time is with hours from 0 to 23.

Both the current time zone and daylight savings time are factored into the calculation of **pa_local()**, and that is the preferred method to find local time.

7.11 Internationalization

The current language in use on the host count be found with **pa_languages()**, which gives a string corresponding to the English name of the language. The languages used are according to the ISO 639-1 standard:

#	Name	#	Name	#	Name	#	Name
1	Afan	36	French	71	Lithuanian	106	Siswati
2	Abkhazian	37	Frisian	72	Macedonian	107	Slovak
3	Afar	38	Galician	73	Malagasy	108	Slovenian
4	Afrikaans	39	Georgian	74	Malay	109	Somali
5	Albanian	40	German	75	Malayalam	110	Spanish
6	Amharic	41	Greek	76	Maltese	111	Sudanese
7	Arabic	42	Greenlandic	77	Maori	112	Swahili
8	Armenian	43	Guarani	78	Marathi	113	Swedish
9	Assamese	44	Gujarati	79	Moldavian	114	Tagalog
0	Aymara	45	Hausa	80	Mongolian	115	Tajik
11	Azerbaijani	46	Hebrew	81	Nauru	116	Tamil
12	Bashkir	47	Hindi	82	Nepali	117	Tatar
13	Basque	48	Hungarian	83	Norwegian	118	Tegulu
14	Bengali	49	Icelandic	84	Occitan	119	Thai
15	Bhutani	50	Indonesian	85	Oriya	120	Tibetan
16	Bihari	51	Interlingua	86	Pashto	121	Tigrinya
17	Bislama	52	Interlingue	87	Persian	122	Tonga
18	Breton	53	Inupiak	88	Polish	123	Tsonga
19	Bulgarian	54	Inuktitut	89	Portuguese	124	Turkish
20	Burmese	55	Irish	90	Punjabi	125	Turkmen
21	Byelorussian	56	Italian	91	Quechua	126	Twi
22	Cambodian	57	Japanese	92	Rhaeto-Romance	127	Uigur
23	Catalan	58	Javanese	93	Romanian	128	Ukrainian
24	Chinese	59	Kannada	94	Russian	129	Urdu
25	Corsican	60	Kashmiri	95	Samoan	130	Uzbek
26	Croatian	61	Kazakh	96	Sangro	131	Vietnamese
27	Czech	62	Kinyarwanda	97	Sanskrit	132	Volapuk
28	Danish	63	Kirghiz	98	ScotsGaelic	133	Welch
29	Dutch	64	Kirundi	99	Serbian	134	Wolof
30	English	65	Korean	100	Serbo-Croatian	135	Xhosa
31	Esperanto	66	Kurdish	101	Sesotho	136	Yiddish
32	Estonian	67	Laothian	102	Setswana	137	Yoruba
33	Faeroese	68	Latin	103	Shona	138	Zhuang
34	Fiji	69	Latvian	104	Sindhi	139	Zulu
35	Finnish	70	Lingala	105	Singhalese		

The ordinal number of the language is given by **pa_language()**, which returns the number of the language from the table above. The ISO 639-1 standard does not specify language ordinal numbers, these are specific to **services**. Note that even if the languages are added to or subtracted to in future implementations, the ordinal numbers will not be changed.

The decimal point character in use can be found with **pa_decimal()**. The current number separator in use is defined with **pa_numbersep()**.

The time and date formats can be derived the country of the host. The main difference between formats is the order of the elements in time and date. The functions **pa_timeorder()** and **pa_dateorder()** give the ordering:

dateorder Code	Date format
1	year-month-day (ISO 8601 standard format)

2	year-day-month
3	month-day-year
4	month-year-day
5	day-month-year
6	day-year-month

timeorder Code	Time format
1	Hour:minute:second (ISO 8601 standard format.
2	Hour:second:minute
3	Minute:hour:second
4	Minute:second:hour
5	Second:hour:minute
6	Second:minute:hour

The separator character for fields in the date is given by **pa_datesep()**, which is '-' in ISO 8601 date formats. The separator character for fields in time is given by **pa_timesep()**, which is ':' in ISO 8601 time formats.

The number of digits in each section of the time and date formats is:

Section	Digits
Year	4
Month	2
Day	2
Hour	2
Minute	2
Second	2

If the time and date format is not set, it defaults to the ISO 8601 standard for time and date formatting. This is the correct format for output that can be read across international boundaries.

Note that the procedures **pa_times()**, **pa_dates()**, **pa_writetime()** and **pa_writedate()** automatically use the internationalization settings of the host to arrive at the host computers natural time and date formatting.

The symbol for the currency used in the country of host is given by **pa_currchr()**.

7.12 Executing Other Programs

An external program can be executed by **pa_exec()**, which takes the command line for the program, including the program name, and all of its parameters and other options on the same line after one or more spaces:

cmd parameter parameter... parameter

This is passed as a string to the **pa_exec()** routine. When programs are executed this way, the executing program does not need to await the finish of the program, nor can it find out if the program ran correctly. If this is required, the routine **pa_execw()** is used. **pa_execw()** will wait for the

program to complete, then place the error return for the program in a variable. This variable will be 0 if the program ran correctly, or non-zero if it didn't. The exact numerical meaning of the error is up to the program executed.

If the system cannot execute programs in parallel, **pa_exec()** is equivalent to **pa_execw()**, but without the return code.

If the environment is to be set for the executed program, the call **pa_exece()** can be used, which takes an environment list. This allows the environment to be retrieved from the current environment or created as new, modified or added to as needed, then passed to the executed program. **pa_execew()** does the same thing, but waits for the program to finish, and returns an error code.

If a command line concept does not exist on the target system, Petit-Ami can pass it via another means, such as a file or environmental variable. Alternately, it could simply be ignored.

7.13 Threads and thread management

A program as executed by **pa_exec**() exists as a complete process. A process is a thread of execution, along with the data that the thread executes or uses. This includes the program code, data, variables, dynamic space, etc. It also includes environment variables and a set of open files.

A process can have any number of threads, even though it starts with only one. **services** provides the means to create more than one thread, to coordinate their execution, and to signal events between threads. Threads can execute anywhere in the code that the main/process thread can, and can access the same data as the other threads in the process. This means that the threads must be coordinated with each other so that data corruption cannot occur.

Threads can run by “time sharing” the CPU between the threads, or they can be done by giving a thread its own CPU or CPU core to run, or a thread can be a combination of the two. It is even possible to move a thread between these two modes dynamically. What is important to know is how threads are implemented is entirely transparent to the programs using **services**. It is also possible that the performance of a program can be enhanced by breaking program work into multiple threads running on multiple concurrent CPUs.

For each thread there exists a thread logical id, from 1 to N where N is the maximum number of threads supported, usually 100. The exact logical number of the thread is choosen for you, and thus allocation of logical thread ids is performed by **services**.

There is one reserved thread number, which is thread number 1, which is the main thread running the process. It may or may not be possible to kill this thread, which is implementation defined. It is also implementation defined as to what happens if the main thread is killed. It may terminate the program, or the program may run until all threads are terminated. For maximum portability, it is recommended that the “all threads are terminated” model be assumed by **services** users.

A new thread is created by **newthread(threadmain)**. This function creates a thread and returns the logical id of the thread. The thread begins by executing the function “threadmain”, which is of the form:

```
void threadmain(void);
```

The created thread will run until either:

1. The threadmain function exits.
2. The thread is killed.

There are many methods by which the thread can communicate with other threads, including the main thread.

Threads can be signaled to terminate by various means. **services** does not provide a method to kill one thread by another thread because this action is inherently insecure. If another thread must be killed, it usually indicates that the application is in error, and the best procedure is to let the thread be killed along with the process when it terminates.

Thread numbers can be reused after the thread terminates.

7.13.1 Concurrency Locks

Concurrency locks are the basic means to coordinate accesses between threads. The fundamental rule of threading is that concurrency locks must be used whenever two different threads can access the same variable data[3].

As with threads, locks are controlled by a logical id, from 1 to N, where N is usually 100. **services** assigns the logical numbers for you.

A concurrency lock is created by **pa_initlock()**. It returns the logical lock number. A lock is destroyed by **pa_deinitlock(ln)**, which takes **ln** as the logical lock number returned by **pa_initlock()**. The lock entry is freed, and the logical lock number can be reassigned by **initlock()**.

A concurrency lock is acquired by **pa_lock(ln)**, where **ln** is the logical id of the lock. It is released by **pa_unlock(ln)**, again where **ln** is the logical lock id. When a lock is acquired, all other threads attempting to acquire the lock are suspended until the lock is released. Generally this is done by "fairlocking", which means that the locks are given out in the order the threads arrive.

When locking, it is possible for threads to deadlock if multiple locks need to be acquired. A typical deadlock situation is:

Thread A:

```
pa_lock(lock_a);
pa_lock(lock_b);
```

Thread B:

```
pa_lock(lock_b);
pa_lock(lock_a);
```

If thread A takes lock_a and thread B takes lock_b, then both threads will deadlock awaiting each other.

A simple way to avoid deadlock is to always take locks in order. A better way is to use hierarchal locking. Using this method, we have a master lock:

Thread A:

[3] In addition, the compiler must be told of parallel access possibilities via the "volatile" keyword.

```
pa_lock(lock_m);
pa_lock(lock_a);
pa_lock(lock_b);
```

Thread B:

```
pa_lock(lock_m);
pa_lock(lock_b);
pa_lock(lock_a);
```

The master lock m is taken any time both A and B must be both acquired. What is nice about this scheme is that it can be used with individual lock requests (lock a and lock b) at the same time. It also can be implemented with simple lock primitives.

7.13.2 Semaphores

The main advantage of threads is that they can share any and all of the data of a process. This means that they can communicate using any data in memory, of any size. However, threads need to know when other threads have finished arranging data. Semaphores are able to send simple signals between threads. A semaphore is like a doorbell. It sends an alert between two threads, but what the exact meaning of the doorbell sounding is up to the threads that are communicating to determine.

Like threads and locks, semaphores have logical identifiers from 1 to N, where N is usually 100. **services** assigns the logical identifiers for you.

Semaphores are tied to locks, because for a thread to determine unambiguously what the meaning of a signal is, must be done within a locked section. A thread looks at data that communicates between threads within a locked region, then accepts a signal that the data is updated while in the locked region.

A signal is created by **pa_initsig()**. It returns the logical signal number. When a signal is no longer needed it is released by **pa_deinitsig**(**sn**), where **sn** is the logical signal number. The signal is sent by **pa_sendsig(sn)**, where **sn** is the logical signal number. Any number of threads can wait for a given signal, and **pa_sendsig()** releases one or more of them to run and process the signal. The threads released must be in a runnable state.

Because usually only one thread can use a signal at a time, the **pa_sendsigone(sn)** call can be used, where **sn** is the logical signal number. As with locks, ideally the first thread to wait on the signal is the one released to run. It is also possible that the underlying system treats **pa_sendsigone()** as equivalent to **pa_sendsig()**, and will wake up one or more threads.

Threads wait on signals using **pa_waitsig(ln, sn)**, where **ln** is a logical lock identifier, and **sn** is a logical signal number. **pa_waitsig()** must be called within the lock **ln**. It releases the lock, waits for the signal, and then reasserts the lock and continues. The lock is released synchronously with checking for a signal.

The typical flow for signaling is:

```
pa_lock(l);
while (<check data condition is true>)
    pa_waitsignal(l, s);
<perform data service>
```

```
pa_unlock(l);
```

7.14 Functions and procedures in services

void pa_list(char* fn, pa_filrec **l);

Form a file list from the filename **fn**, and return in the file entry list **l**. The filename **fn** may contain wildcards, and may be fully pathed, or refer to the current directory. If no files are found, then the list pointer is returned NULL.

void pa_times(char* s, int sl, int t);

Place time from S2000 time **t** in string **s**, with maximum length **sl**.

void pa_dates(char* s, int sl, int t);

Place date from S2000 time **t** in string **s**, with maximum length **sl**.

void pa_writetime(FILE *f, int t);

Write time from S2000 time **t** to text file **f**.

void pa_writedate(FILE *f, int t);

Write date from S2000 time **t** to text file **f**.

long pa_time(void);

Returns current S2000 time in GMT

long pa_local(long t);

Converts the given GMT S2000 time **t** to local time, using the time zone offset and daylight savings status in the host computer, and returns the result.

long pa_clock(void);

Returns 100 microsecond, free running time.

long pa_elapsed(long r);

Given a stored **clock** time **r**, will check the current **clock** time and find the total number of 100 microsecond ticks since the given time, then return that. Accounts for timer wraparound.

int pa_validfile(char* s);

Parses and checks the file specification **s** for a valid filename on the current system. Returns 1 if valid, otherwise 0.

int pa_validpath(char* s);

Parses and checks the file specification **s** for a valid path on the current system. Returns 1 if valid, otherwise 0.

int pa_wild(char* s);

Checks if the file specification **s** contains wildcards. Returns 1 if so, otherwise 0.

void pa_getenv(char* ls, char* ds, int dsl);

Finds and returns an environment string. **ls** contains the name of the string to look up, **ds** with maximum length **dsl** contains the resulting string as found. If there is no environment string by that name, the return is either empty, or NULL.

void pa_setenv(char* sn, char* sd);

Finds and sets an environment string. **sn** contains the string name to set, and **sd** contains the contents to set it to. If there is no string by that name, it is created, otherwise the old string is replaced.

void pa_allenv(pa_envrec **el);

Returns a complete list of the strings in the environment to **el**.

void pa_remenv(char* sn);

Remove a string **sn** from the environment. The string is found by name, and removed from the environment. No error results if the string does not exist.

void pa_exec(char* cmd);

Execute external program, with parameters, from the string **cmd**. Does not wait for the program to finish, and cannot detect if it finished with an error.

void pa_exece(char* cmd, pa_envrec *el);

Execute external program, with parameters from the string **cmd** and full environment. The environment is passed as a list in **el**. Does not wait for the program to finish, and cannot detect if it finished with an error.

void pa_execw(char* cmd, int *e);

Execute external program, with parameters from the string **cmd**. Waits for the program to finish, and returns its error code in **e**. The error code is 0 for no error, otherwise the error is a code specified by the program executed.

void pa_execew(char* cmd, pa_envrec *el, int *e);

Execute external program, with parameters from the string **cmd** and full environment. The environment is passed as a list in **el**. Waits for the program to finish, and returns its error code in **e**. The error code is 0 for no error, otherwise the error is a code specified by the program executed.

void pa_getcur(char* fn, int fnl);

Get current path to **fn** with maximum length **fnl**.

void pa_setcur(char* fn);

Set current path from string **fn**. Sets the default path for all file specifications.

void pa_getpgm(char* p, int pl);

Get the program path to **p** with maximum length pl. Returns the program path, which is the path the program running was loaded from.

void pa_getusr(char* fn, int fnl);

Get user path to **fn** or returns it. Return the user path, which is a path specific to each user.

void pa_brknam(char* fn, char* p, int pl, char* n, int nl, char* e, int el);

Break down file specification. Breaks the file specification **fn** down into path **p**, name **n**, and extension **e**, with maximum lengths **pl**, **nl**, and **el**. Note that any one of the resulting components could be blank, if it does not exist in the name.

void pa_maknam(char* fn, int fnl, char* p, char* n, char* e);

Create file specification from components. Creates file specification **fn** from path **p**, name **n**, and extension **e**. Components may be blank, but the path and the name cannot both be blank.

void pa_fulnam(char* fn, int fnl);

Create full file specification from a partial file specification **fn**. Given a file specification with an incomplete path (either by using the default path, or mnemonic shortcuts for things like parent directory), creates a fully pathed name of standard form. This can "normalize" file specifications, for comparisons, and to store the complete path for the file. The fully pathed result is returned in **fn**, with maximum length **fnl**.

void pa_setatr(char* fn, pa_attrset a);

Set attributes. Given a file by name **fn**, the attributes in the set **a** are set true for the file.

void pa_resatr(char* fn, pa_attrset a);

Reset attributes. Given a file by name **fn**, the attributes in the set **a** are set false for the file.

void pa_bakupd(char* fn);

Set backup time current. Given a file by name **fn**, sets the backup time for the file as current. Backup programs should also reset the archive bit to show that backup has occurred.

void pa_setuper(char* fn, pa_permset p);

Set user permissions. Given a file by name **fn**, the permissions in the set **p** are set true for the file.

void pa_resuper(char* fn, pa_permset p);

Reset user permissions. Given a file by name **fn**, the permissions in the set **p** are set false for the file.

void pa_setgper(char* fn, pa_permset p);

Set group permissions. Given a file by name **fn**, the permissions in the set **p** are set true for the file.

void pa_resgper(char* fn, pa_permset p);

Reset group permissions. Given a file by name **fn**, the permissions in the set **p** are set false for the file.

void pa_setoper(char* fn, pa_permset p);

Set other permissions. Given a file by name **fn**, the permissions in the set **p** are set true for the file.

void pa_resoper(char* fn, pa_permset p);

Reset group permissions. Given a file by name **fn**, the permissions in the set **p** are set false for the file.

void pa_makpth(char* fn);

Make path using **fn**. Creates a new path or directory. If the path already exists, it's an error.

void pa_rempth(char* fn);

Remove path **fn**. Removes a path, or directory. The directory must exist, and must be empty of any files or other directories, or an error results.

void pa_filchr(pa_chrset fc);

Returns the set of valid filename characters in **fc**.

char pa_optchr(void);

Find the option character. Returns the character that is used to introduce options in command liens on the current system.

char pa_pthchr(void);

Returns the character used to separate components in a path in the current system.

int pa_latitude(void);

Returns the latitude of the current location as a binary number, where 0 is the equator and INT_MAX is the north pole, and –INT_MAX is the south pole.

If pa_longitude is –INT_MAX, indicating that the position is invalid, then pa_latitude is also invalid.

int pa_longitude(void);

Returns the longitude of the current location as a binary number, where 0 is the prime meridian (Greenwich, England), and INT_MAX is the other side of the world eastward, and –INT_MAX is the other side of the world, westward.

If the current position is not available, then –INT_MAX is returned. Note that the back side of the planet is already indicated by INT_MAX.

int pa_altitude(void);

Returns the binary height above the earth's normalized surface, from 0 (ground) to 100km or space at INT_MAX. Returns –INT_MAX if the altitude is not available.

int pa_country(void);

Returns the ISO 3166-1 numeric code for the current country..

void pa_countrys(char* s, int sl, int c);

Returns the string corresponding to the current country. This is an empty string if no position is known. The resulting string must fit into the string **s** in the space given as **sl**.

int pa_timezone(void);

Returns the offset, in seconds, of the current time zone from GMT, in the range of -12 to +14 hours.

int pa_daysave(void);

Returns 1 if daylight savings is in effect at the current location, otherwise 0.

int pa_time24hour(void);

Returns 1 if 24 hour time is in effect at the current location, otherwise 0.

int pa_language(void);

Returns the ordinal number of the language in effect for the current location.

void pa_languages(char* s, int sl, int l);

Returns the name of the language in use for the current location. The name string is placed at s, which is a buffer with length l. The buffer must be long enough to contain the full name plus a terminating zero.

char pa_decimal(void);

Returns the character used to separate the fractional part from the rest of numbers (the decimal point).

char pa_numbersep(void);

Returns the character used to separate parts of a number, every 3 digits.

int pa_timeorder(void);

Returns the time order code used to specify the local time format in use. See the table for valid codes.

int pa_dateorder(void);

Returns the the date order code used to specify the local time format in use. See the table for valid codes.

char pa_datesep(void);

Returns the character used to separate date components in the current system.

char pa_timesep(void);

Returns the character used to separate time components in the current system.

char pa_currchr(void);

Returns the character used to separate components in a path in the current system.

int pa_newthread(void *(*threadmain)(void *));

Start new thread. Expects a function, **threadmain**, to start as the new thread. A logical thread number is created from 1 to N, where N is the maximum number of threads supported. The new thread number is allocated and returned to the caller. The thread is created and executes at the given function. The thread will run until it returns from the called function or is killed.

int pa_initlock(void);

Creates a new concurrency lock and returns the logical id for it.

int pa_deinitlock(int ln);

Releases a concurrency lock by logical id **ln**.

void pa_lock(int ln);

The given lock by logical id **ln** is aquired. Generally fairlocking can be assumed, which means that for N waiters on the lock, they will be given the lock first come first served.

void pa_unlock(int ln);

The concurrency lock by logical id **ln** is released. The next thread queued for the lock and that is in a runnable state is set to run.

int pa_initsig(void);

Creates a new concurrency signal and returns the logical id for it.

int pa_deinitsig(int sn);

Releases a concurrency lock by logical id **sn**.

void pa_sendsig(int sn);

Returns the character used to separate components in a path in the current system. Flags an event to the given signal by logical id **sn**. Either all waiters on the signal or just one is set to run by a signal.

pa_sendsigone(int sn);

Returns the character used to separate components in a path in the current system. Flags an event to the given signal by logical id **sn**. Only one thread is signaled to run. This version of signal is used when only one waiter can use the event signaled.

Note that it is possible for this call to be equivalent to **pa_signal()** on a given implementation. Thus programs should check if the signaled event is still active, and not just assume it.

void pa_waitsig(int ln, int sn);

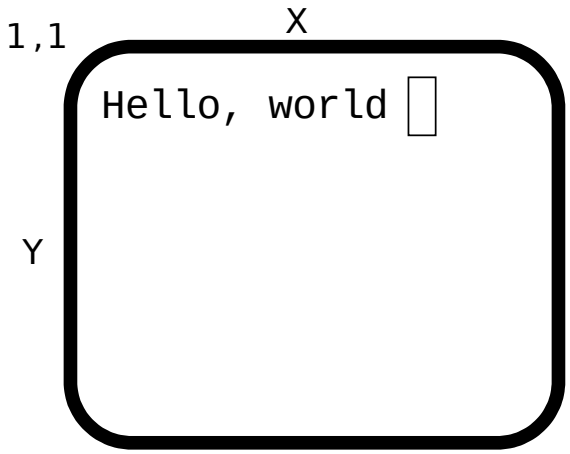
Wait for a given signal. Accepts a currency lock logical id **ln**, and a signal logical id **sn**. Releases the concurrency lock and waits for the given signal, then returns when the signal flags.

The purpose of accepting a lock number is that it is released and the signal is checked atomically. This means other threads that are not signaling will not run, and thus the wait and signal operations are synchronized together.

8 Terminal Interface Library

Standard ANSI C uses an I/O paradigm that is serial, or more correctly "line oriented". Each line is built up in sections, then output to the I/O device with an appended "end of line". The end of line causes the current line to be completed, and the next line begins.

The next level of paradigm is the presentation of lines onto a 2d text surface. This can simply be an emulation of the serial only system or “virtual paper” using a screen that scrolls up from the bottom. The next step is to allow full addressing of the 2d surface and allowing text to be placed anywhere within that surface. To this is added colors, different text presentation modes, and finally advanced, multiple device input.



terminal starts in ANSI C compatible mode, then allows the program to move to a full addressable surface without automatic scrolling. In advanced mode, **terminal** emulates an infinite virtual surface where the terminal exists as a window clipped to the origin. This is the most consistent model of such a surface.

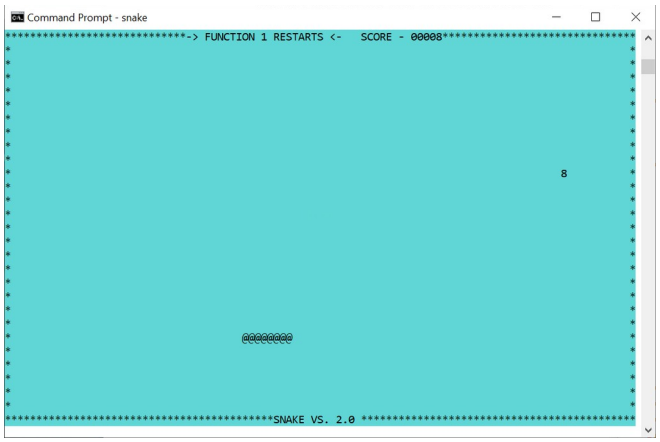
Because **terminal** overrides the interface between the program and the operating system it runs on, all of the ANSI C defined serial I/O works compatibly with the **terminal** presented surface. It is also modal, meaning that changes to character modes affect all further output to the terminal.

terminal introduces the concept that several devices can be used for input at the same time. The normal user keyboard is supplemented by a mouse, timers and a joystick. These are all implemented via the event concept, which unifies the input and removes the need to poll multiple devices.

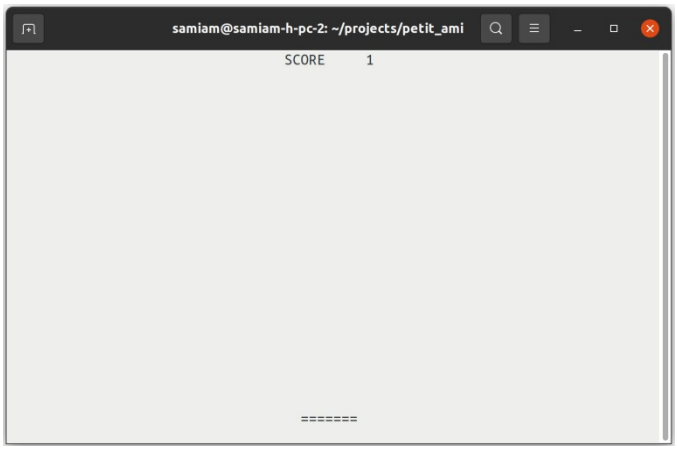
terminal gives a set of logical events for common control keys from the keyboard that allow the program to avoid direct recognition of implementation dependent key codes.

In addition to the default presentation surface, **terminal** is capable of switching between multiple display surfaces. This capability has several uses.

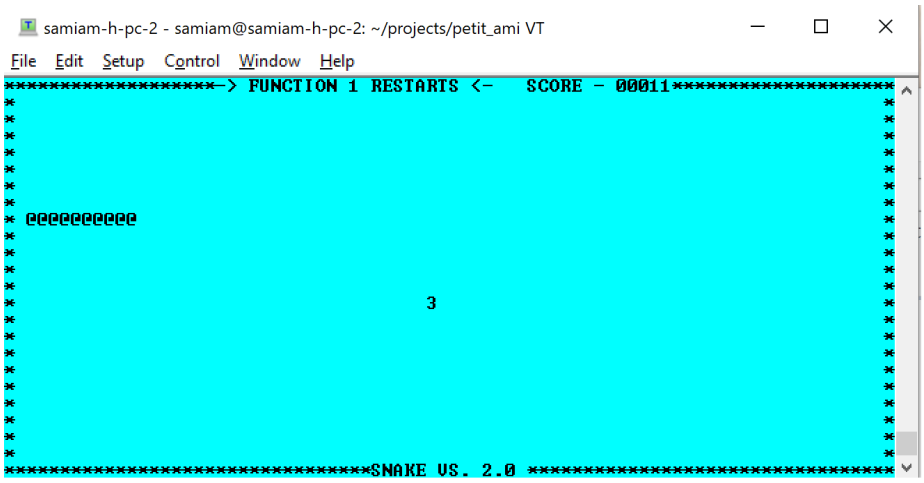
Examples:



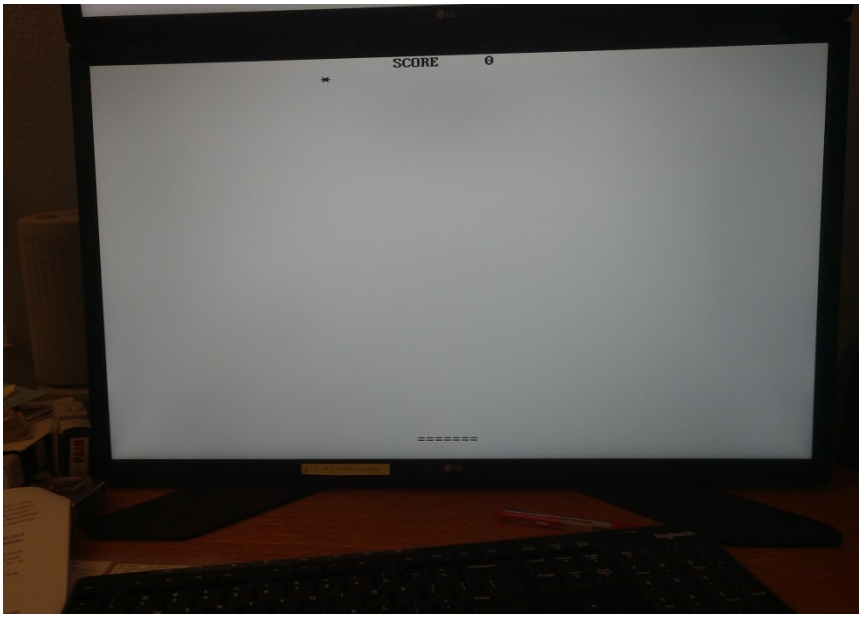
Terminal programs can appear in any console or terminal window, on multiple operating systems, or even on an actual terminal. Here on a Windows console.



Here terminal is on a Linux xterm.



And here remoted by ssh from Linux to TeraTerm on Windows. The xterm escape character protocol makes an excellent lightweight remote serial display protocol.



Playing pong on a Linux CLI (no Xwindows display). xterm is implemented directly to the console.

These examples only scratch the surface. xterm can be used via an RS232 or other serial link, which could go to a remote system or embedded system.

8.1 ANSI C Compatible Mode

To write data to the **terminal** screen, ANSI C I/O calls are used. For example, a **putchar(c)** statement places each character on the screen, then moves the cursor to the right, obeying any automatic line wrapping. If a **putchar(c)** is called at the end of the line, then the cursor is returned to the left side of the screen, one line down. If the end of the line is reached, and automatic scrolling is enabled, then the screen will scroll upwards.

The ‘\f’ output character, which causes a printer to move to the next (blank) page, is emulated by clearing the screen and moving the cursor to home position at (1,1).

All of the ANSI C I/O calls are supported in **terminal** mode, including **printf(f,...)**, **fprintf(f,...)**, **fscanf(f,...)**, **fputc(c,f)**, **fgetc(f)**, etc.

8.2 Specifing the Main Window

All calls in **terminal** take a file to specify which window is supplying input or getting output, typically **stdin** or **stdout**. This is for upward compatibility reasons. For the majority of terminal calls, **stdout** is used, except for **pa_event()**, which gets input from **stdin**.

8.3 Basic Cursor Positioning

The cursor is the point where text is entered onto the screen. It’s usually marked with a blinking block or underline. To move the cursor one character up, down, left or right, the **pa_up(f)**, **down(f)**, **pa_left(f)** and **pa_right(f)** procedures are used. To move the cursor anywhere on the screen, the **pa_cursor(f,x,y)** call is used. To find out where the cursor currently is, the functions **pa_curx(f)** and **pa_cury(f)** return the current x and y coordinates of it.

The character cells on the screen are labeled from 1 to N, where in x 1 is the left side of the screen, and N is the right side of the screen. In y, 1 is the top of the screen, and N is the bottom of the screen.

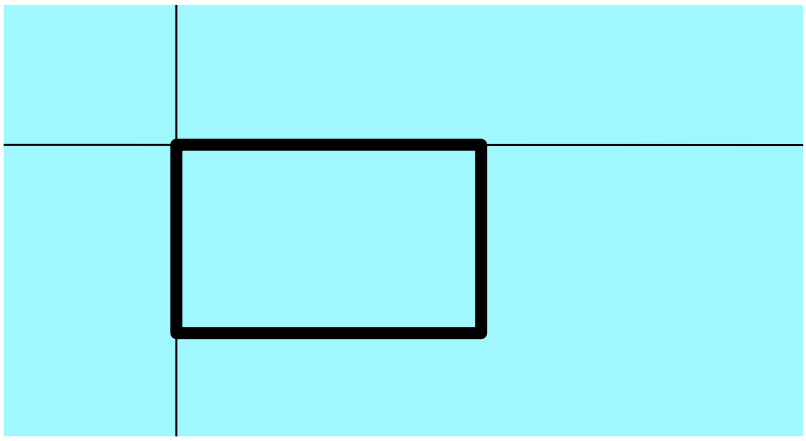
The actual size of the screen is system specific, and can be found by **pa_maxx(f)** and **pa_maxy(f)**, which return the maximum index in x and y for the screen. When a terminal is emulated in a windowing system, it usually by default 25 lines of 80 characters each, because that was a very common size in terminals.

The cursor can be moved to the home, or 1,1 position, by the **pa_home(f)** procedure.

8.4 Automatic Mode

Automatic mode is the mode that causes **terminal** to scroll upwards when the bottom of the screen is reached, and a newline character (‘\n’) is written. Line wrap, or the wrapping of characters back to the left at the next line if text is written off the right hand side, is an automatic mode. The automatic mode is useful for emulating ANSI C serial mode programs, but rapidly gets in the way for advanced programs.

Automatic mode can be turned off with **pa_auto(f,e)**. Turning **pa_auto(f,e)** off converts (or actually, reveals) the screen as a character surface that goes from –**INT_MAX** to +**INT_MAX** in both x and y, and has its origin at 1,1, and the screen is a "viewport" on this surface that extends from 1,1 to **pa_maxx(f)**, **pa_maxy(f)**. Text can be drawn anywhere, including off screen, but the characters outside of the 1,1 to **pa_maxx(f)**, **pa_maxy(f)** box will not be seen, and are "clipped out" of view. It can be determined if the cursor lies within the screen's bounds by **pa_curbnd(f)**.



The "virtual screen" that appears with **pa_auto(f,e)** off is a good match for today's windowing environments. Text can be drawn without worrying if it will cause the line to wrap, or the screen to scroll. And if a line of text happens to extend off the screen, that does not cause an error.

8.5 Tabbing

terminal will keep track of tabs set in x, for any character position. When **terminal** starts, the tabs are set to every 8th position on the screen, i.e., positions x= 9, 17, etc.

Outputting a tab character will cause the cursor to move to the next tab position on the line. If the cursor is at a tab position, then it will move to the next one. Tab positions can be set by **pa_settab(f,t)**, and cleared by **pa_restab(f,t)**. **pa_clrtab(f)** clears all set tabs.

8.6 Scrolling

The screen can be scrolled by the **pa_scroll(f,x,y)** procedure, which implements arbitrary direction scrolling. If the **x** value given is positive, then the screen data scrolls up. If the **x** value is negative, then the screen data scrolls down. If the **y** value is positive, the screen data scrolls left, and if it's negative, the screen data scrolls right. If either **x** or **y** is 0, then there is no movement in that direction.

Value	Cursor Direction	Screen data
+x	Down	Moves up
-x	Up	Moves down
+y	Right	Moves left
-y	Left	Moves right

8.7 Colors

There are two colors to set for text. One is the foreground, and the other is the background. The foreground is the color within the character itself. The background is the space behind the character. The possible colors are chosen from the two sets of primary colors:

```
type pa_color = (pa_black, pa_white, pa_red, pa_green,
                 pa_blue,pa_cyan, yellow, magenta);
```

Characters are written in the currently set foreground and background colors. The foreground color is set by **pa_fcolor(f,c)**, and the background color by **pa_bcolor(f,c)**.

The current background color is also used to set the color of any blanked out areas caused by other commands. For example, outputting '\f' (form feed) clears the screen to the background color. Scrolls, either programmer selected or automatic, use the background color for any uncovered areas that are blanked out.

8.8 Attributes

terminal provides many attribute controls, each of which can be switched on or off individually.

Attribute	Result
Reverse	Enables reverse video.
underline	Underlines each character.
superscript	Gives a smaller and higher character for subscripting.
subscript	Gives a smaller and lower character for superscripting.
italic	Prints in italic or slanted characters.
bold	Prints in extra dark, or bold characters.
Strikeout	Prints characters with a horizontal bar through them.
blink	Blinks each character on and off.

For programs to maintain portability, the programmer should assume two things. First, that only a single attribute at one time can be set. This would mean that setting a second attribute after a first one is set, would cause the first attribute to be unset.

Second, the programmer should not assume that any one particular attribute is available. Many older terminals do have more than one or two attributes. **pa_superscript(f,e)**, **pa_subscript(f,e)**, **pa_italic(f,e)** and **pa_strikeout(f,e)** are rare in standalone terminals. **pa_superscript(f,e)**, **pa_subscript(f,e)**, **pa_italic(f,e)** and **pa_bold(f,e)** are rare today on graphical windowing systems because they alter the geometry of the characters such that it is difficult or impossible to emulate a character cell in such a system. **pa_blink(f,e)** is also rare in graphical systems because of the computational work required.

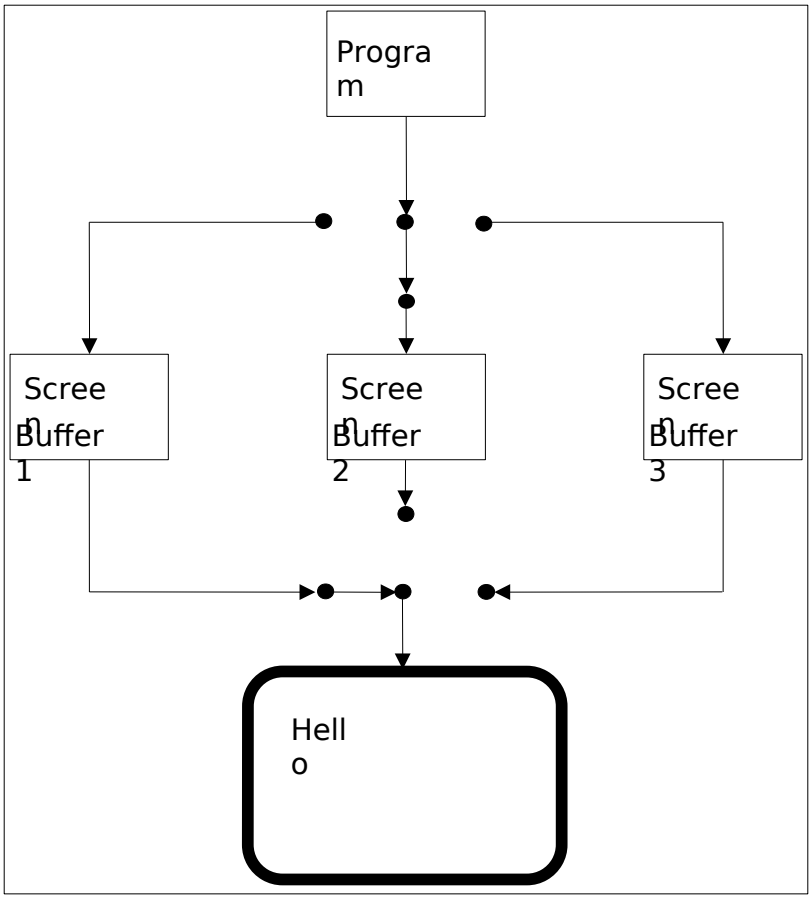
For this reason, there is a general mode, **pa_standout(f,e)**, that can be enabled or disabled just like an attribute. What **pa_standout(f,e)** does is enable an attribute the terminal does have, so that the program does not have to select a specific attribute. Most commonly this is implemented as reverse video.

8.9 Multiple Surface Buffering

terminal implements the ability to have logical screens in buffers. This is a copy of the characters on the screen, saved in a buffer of the same size as the screen. Logical buffers have many uses. Multiple screens can be kept, and quickly switched to the user. A program, like an editor, that wants to save the screen as it was before it started, can switch away to a second screen to do its work, and then switch back to the original screen to restore its contents. Buffers can also be used to accomplish animation.

The **select(u,d)** procedure sets both the current screen to be displayed, and also the screen that updates are to go to. When **terminal** starts, these are both set to the same screen, number 1. Any screen can be selected, and the display screen and the update screen can be different. If the update screen is not the one

in display, then all of the write statements will place characters in the buffer, but not on screen. This "split mode" is typically used for animation.



Implementations are free to limit the total number of logical screens available. The user should assume no more than 10 logical screens are available.

8.10 Advanced Input

terminal implements an advanced output model that is upward compatible with the ANSI C serial model. Similarly, an advanced input model is also implemented that meets the needs of newer systems.

Today's systems deal with multiple input devices, not just the keyboard. Devices include a mouse, a joystick, timers, and other future devices. The standard method on small computers was to poll for such devices, or accept interrupts from them. The difficulty with those solutions was that these methods did not fit well with modern multitasking systems.

One method would be to use a multitask thread per device. However, this complicates small programs unnecessarily. The common solution today is "events", and the "event loop". The event model takes advantage of the fact that user input devices don't generate a lot of high bandwidth data. Each of the devices attached to the input from the user have their data packaged as "messages", which are small

records, that are a description of the event. These events are then placed in a queue, and the program pulls the events, one at a time, from the queue and acts on them.

The description of an event record is:

```
/** events */
typedef enum {

    /** ANSI character returned */     pa_etchar,
    /** cursor up one line */          pa_etup,
    /** down one line */               pa_etdown,
    /** left one character */          pa_etleft,
    /** right one character */         pa_etright,
    /** left one word */               pa_etleftw,
    /** right one word */              pa_etrightw,
    /** home of document */            pa_ethome,
    /** home of screen */              pa_ethomes,
    /** home of line */                pa_ethomel,
    /** end of document */             pa_etend,
    /** end of screen */               pa_etends,
    /** end of line */                 pa_etendl,
    /** scroll left one character */   pa_etscrl,
    /** scroll right one character */  pa_etscrr,
    /** scroll up one line */          pa_etscru,
    /** scroll down one line */        pa_etscrd,
    /** page down */                   pa_etpagd,
    /** page up */                     pa_etpagu,
    /** tab */                         pa_ettab,
    /** enter line */                  pa_etenter,
    /** insert block */                pa_etinsert,
    /** insert line */                 pa_etinsertl,
    /** insert toggle */               pa_etinsertt,
    /** delete block */                pa_etdel,
    /** delete line */                 pa_etdell,
    /** delete character forward */    pa_etdelcf,
    /** delete character backward */   pa_etdelcb,
    /** copy block */                  pa_etcopy,
    /** copy line */                   pa_etcopyl,
    /** cancel current operation */    pa_etcan,
    /** stop current operation */      pa_etstop,
    /** continue current operation */  pa_etcont,
    /** print document */              pa_etprint,
    /** print block */                 pa_etprintb,
    /** print screen */                pa_etprints,
    /** function key */                pa_etfun,
    /** display menu */                pa_etmenu,
    /** mouse button assertion */      pa_etmouba,
    /** mouse button deassertion */    pa_etmoubd,
    /** mouse move */                  pa_etmoumov,
    /** timer matures */               pa_ettim,
    /** joystick button assertion */   pa_etjoyba,
    /** joystick button deassertion */ pa_etjoybd,
    /** joystick move */               pa_etjoymov,
    /** window was resized */          pa_etresize,
```

```
    /** terminate program */           pa_etterm,
    /** frame sync */                  pa_etframe

} pa_evtcod;

/** event record */

typedef struct {

    /* identifier of window for event */ int winid;
    /* event type */                      pa_evtcod etype;
    /* event was handled */               int handled;
    union {

        /* these events require parameter data */

        /** etchar: ANSI character returned */  char echar;
        /** ettim: timer handle that matured */ int timnum;
        /** etmoumov: */
        struct {

            /** mouse number */   int mmoun;
            /** mouse movement */ int moupx, moupy;

        };
        /** etmouba */
        struct {

            /** mouse handle */  int amoun;
            /** button number */ int amoubn;

        };
        /** etmoubd */
        struct {

            /** mouse handle */  int dmoun;
            /** button number */ int dmoubn;

        };
        /** etjoyba */
        struct {

            /** joystick number */ int ajoyn;
            /** button number */   int ajoybn;

        };
        /** etjoybd */
        struct {
```

```
            /** joystick number */ int djoyn;
            /** button number */   int djoybn;

        };
        /** etjoymov */
        struct {

            /** joystick number */      int mjoyn;
            /** joystick coordinates */ int joypx, joypy, joypz;

        };
        /** function key */ int fkey;

    };

} pa_evtrec, *pa_evtptr;
```

The next event for the program is retrieved via the **pa_event(f,er)** call. The basis of the event system is that events must be retrieved often enough that the input queue does not fill up. Thus, an "event model" program will be centered on the event loop that gets the next event, acts on it, and returns to the top for more events.

An important principle of event handling is that events that are not defined for a compliant program are ignored. That is, if an event not defined for **terminal** is received, it will be ignored. This is the basis of upward compatibility. If a new event is defined, existing programs will simply ignore it.

A typical event loop is as follows.

```
#include <stdio>
#include <terminal.h>

main()

{

    pa_evtrec erl

    do {

        pa_event(&er); /* get next event */
        switch (er.etype) {

            case pa_etchar:
                if (er.echar = ‘g’) /* perform character action */;
                break;
            case pa_etmoumov: /* perform mouse move action */
            default: /* do nothing */

        }

    } while (er.etype != pa_etterm);

}
```

Note the final default case that does nothing.

8.11 Advanced Input in C++

The event record in C++ is the same as for C, but the “pa_” prefix is not used. The terminal.hpp file contains a namespace terminal, and the namespace is joined by:

```
using namespace terminal;
```

Alternately, symbols from the terminal namespace can be specified individually:

```
terminal::event(stdin, &er);
```

The description for a C++ event record is:

```
/* events */
typedef enum {

    /** ANSI character returned */     etchar,
    /** cursor up one line */          etup,
    /** down one line */               etdown,
    /** left one character */          etleft,
    /** right one character */         etright,
    /** left one word */               etleftw,
    /** right one word */              etrightw,
    /** home of document */            ethome,
    /** home of screen */              ethomes,
    /** home of line */                ethomel,
    /** end of document */             etend,
    /** end of screen */               etends,
    /** end of line */                 etendl,
    /** scroll left one character */   etscrl,
    /** scroll right one character */  etscrr,
    /** scroll up one line */          etscru,
    /** scroll down one line */        etscrd,
    /** page down */                   etpagd,
    /** page up */                     etpagu,
    /** tab */                         ettab,
    /** enter line */                  etenter,
    /** insert block */                etinsert,
    /** insert line */                 etinsertl,
    /** insert toggle */               etinsertt,
    /** delete block */                etdel,
    /** delete line */                 etdell,
    /** delete character forward */    etdelcf,
    /** delete character backward */   etdelcb,
    /** copy block */                  etcopy,
    /** copy line */                   etcopyl,
    /** cancel current operation */    etcan,
    /** stop current operation */      etstop,
    /** continue current operation */  etcont,
    /** print document */              etprint,
    /** print block */                 etprintb,
    /** print screen */                etprints,
    /** function key */                etfun,
    /** display menu */                etmenu,
    /** mouse button assertion */      etmouba,
    /** mouse button deassertion */    etmoubd,
    /** mouse move */                  etmoumov,
    /** timer matures */               ettim,
    /** joystick button assertion */   etjoyba,
    /** joystick button deassertion */ etjoybd,
    /** joystick move */               etjoymov,
    /** window was resized */          etresize,
```

```
    /** terminate program */           etterm

} evtcod;

typedef struct {

    /* identifier of window for event */ int winid;
    /* event type */                      evtcod etype;
    /* event was handled */               int handled;
    union {

        /* these events require parameter data */

        /** etchar: ANSI character returned */  char echar;
        /** ettim: timer handle that matured */ int timnum;
        /** etmoumov: */
        struct {

            /** mouse number */   int mmoun;
            /** mouse movement */ int moupx, moupy;

        };
        /** etmouba */
        struct {

            /** mouse handle */  int amoun;
            /** button number */ int amoubn;

        };
        /** etmoubd */
        struct {

            /** mouse handle */  int dmoun;
            /** button number */ int dmoubn;

        };
        /** etjoyba */
        struct {

            /** joystick number */ int ajoyn;
            /** button number */   int ajoybn;

        };
        /** etjoybd */
        struct {

            /** joystick number */ int djoyn;
            /** button number */   int djoybn;
```

```
        };
        /** etjoymov */
        struct {

            /** joystick number */      int mjoyn;
            /** joystick coordinates */ int joypx, joypy, joypz;

        };
        /** function key */ int fkey;

    };

} evtrec, *evtptr;
```

8.12 Buffer Follow mode

Some terminal implementations can exist in windowed environments, and that window can be resized. For this reason, terminal provides a mode called “buffer following” that allows a program to resize it’s output to “follow” the window size onscreen. The function **pa_sizbuf(f,x,y)** will change the size of the buffers to the size in x and y. When the buffer is resized, the functions **pa_maxx(f)** and **pa_maxy(f)** will change to reflect the new size. If the buffer is larger than the window size, it will be clipped, and the system may or may not provide a way for the user to examine the contents of the buffer via scrolling or other means. If the buffer is smaller than the window size, it is presented at the origin, and the unoccupied space in the window is cleared to the background color. After a **pa_sizbuf(f,x,y)** call, all of the screen buffers are resized, and any previous contents are lost, and the buffer is cleared to spaces.

When the window size is changed, a **pa_etresize** message will be sent to the window. This message carries the new size of the window. The message won’t be sent on a system that does not perform windowing (a full screen terminal), and the program can ignore it if it does not implement buffer following. To implement buffer following, the program would take the **rszx** and **rszy** size parameters from the message and use the **pa_sizbuf(f,x,y)** function to resize the buffer. The **pa_maxx(f)** and **pa_maxy()** functions will then “follow” the new window size and the program should rewrite the update buffer for all active screens.

The main thing to understand about buffer follow mode is that it is optional. The client program can ignore a change in the onscreen display size, and keep maintaining a buffered virtual screen, or it can change its buffer to follow the screen.

8.13 Focus events

A terminal that is hosted by a windowing system may or may not have focus. Focus indicates the terminal will get input. The focus events only apply if the terminal is within a window on a windowed system. Focus is always true on a non-windowed system.

There are two focus events, **pa_etfocus** and **pa_etnofocus**. **pa_etfocus** indicates the input is active to the terminal. **pa_etnofocus** indicates no input will occur. The events only are sent when the state of focus changes. Typically, the focus state will be accompanied by a change in the cursor.

8.14 Hover events

A terminal that is hosted by a windowing system may or may not have hover. Hover indicates the mouse is over the terminal window. The hoever events only apply if the terminal is within a window on a windowed system. Hover is always true on a non-windowed system.

There are two hover events, pa_ethover and pa_etnohover. pa_ethover indicates the mouse is over the terminal window. pa_etnohover indicates the mouse is no longer over the terminal window. The events areonly sent when the statate of hover changes.

8.15 Legacy Input

terminal mode supports all of the ANSI C input methods (**fgetc(c,f)**, **fscanf(f,...)**, etc.). It does this by calling **pa_event(f,er)** for you, and discarding all events not related to line character input. Typically you want to get your own input with **pa_event(f,er)**. However, the legacy mode can be quite useful for inputting lines of characters from the user, because **terminal** mode often features line editing functions.

8.16 Event callbacks

An alternative to setting up an event loop, or a complement to it, is to arrange callbacks for individual events. Events are overridden with the call **pa_eventover(e, eh, oeh)**. The event to be overridden is specified, along with a function to be called when the event occurs. The old routine that was executed on the callback is also returned, and normally that is executed within the handler routine:

```
/** event function pointer */
typedef void (*pa_pevthan)(pa_evtrec*);

#include <stdio.h>

pa_pevtptr oldhandler; /* previous event handler */

void myevent(pa_evtptr er)

{

    /* perform event handling using event record er */
    oldhandler(er); /* pass back to previous handler */

}

main()

{

    /* override event handler */
    pa_eventover(etchar, myevent, &oldhandler);

}
```

pa_event(f,er) directly calls the overriding function, and the event is processed there. If the overriding procedure does not want to handle the event, then the inherited version of the procedure is called in the overrider. This accomplishes two things. First, any other overriders that are chained to the procedure are executed, so that they may handle the event. Finally, if none of the overriders wish to handle the event, the chain ends with the default implementation, which then flags that the event is to be returned to the caller of **pa_event(f,er)**. In this way, if none of the overriders handle the event, it is returned as a normal record back to the **pa_event(f,er)** caller.

There is no parallel execution implied in such event callbacks. The **pa_event(f,er)** function still must be called to activate the event procedures, and all such procedures run in the context of the current process.

The event override mechanism is a way to break the rigid formalism of event loop design. In the event loop model, the event loop must be changed anytime there is new code that needs to handle events. Using event procedures, new handlers can be added without such modification. This enhances modularity, since the new code may be in a different module. This aids the extendability of the system.

Event procedures are also a way to obfuscate a program by making it less obvious where the flow of control is in the program. The advantage to the event loop model is that it creates clear flow of control, even when handling asynchronous user generated events.

8.17 Event Callbacks in C++

In C++, an alternative to receiving events as structures are the event based virtual methods:

```
/* virtual callbacks */
virtual int evchar(char c);
virtual int evup(void);
virtual int evdown(void);
virtual int evleft(void);
virtual int evright(void);
virtual int evleftw(void);
virtual int evrightw(void);
virtual int evhome(void);
virtual int evhomes(void);
virtual int evhomel(void);
virtual int evend(void);
virtual int evends(void);
virtual int evendl(void);
virtual int evscrl(void);
virtual int evscrr(void);
virtual int evscru(void);
virtual int evscrd(void);
virtual int evpagd(void);
virtual int evpagu(void);
virtual int evtab(void);
virtual int eventer(void);
virtual int evinsert(void);
virtual int evinsertl(void);
virtual int evinsertt(void);
virtual int evdel(void);
virtual int evdell(void);
virtual int evdelcf(void);
virtual int evdelcb(void);
virtual int evcopy(void);
virtual int evcopyl(void);
virtual int evcan(void);
virtual int evstop(void);
virtual int evcont(void);
virtual int evprint(void);
virtual int evprintb(void);
virtual int evprints(void);
virtual int evfun(int k);
virtual int evmenu(void);
virtual int evmouba(int m, int b);
virtual int evmoubd(int m, int b);
virtual int evmoumov(int m, int x, int y);
virtual int evtim(int t);
virtual int evjoyba(int j, int b);
virtual int evjoybd(int j, int b);
virtual int evjoymov(int j, int x, int y, int z);
virtual int evresize(void);
virtual int evterm(void);
```

Each event method corresponds to an event from the **evtrec** structure. When **event()** has an event to return to the calling program, it first calls the default implementation of the corresponding virtual method, which flags that the event is unhandled, and should be returned to the caller.

The alternative method is activated by overriding the virtual procedure corresponding to the event that is to be received directly. **event()** directly calls the overriding procedure, and the event is processed there. If the overriding procedure handles the event, it returns true (non-zero). If the overriding procedure does not want to handle the event, then the inherited version of the procedure is called in the overrider. This accomplishes two things. First, any other overriders that are chained to the procedure are executed, so that they may handle the event. Finally, if none of the overriders wish to handle the event, the chain ends with the default implementation, which then flags that the event is to be returned to the caller of **event()**. In this way, if none of the overriders handle the event, it is returned as a normal record back to the **event()** caller.

There is no parallel execution implied in such event callbacks. The **event()** function still must be called to activate the event procedures, and all such procedures run in the context of the current process.

The event procedures are prefixed with “ev” (event) to prevent them from colliding with the names of the normal **terminal** functions.

This is an example of using event callbacks in C++:

```
#include <stdlib.h>
#include <iostream>

#include <terminal.hpp>

using namespace terminal;

class myterm: public term

{

    public:
    int evterm(void) {

        std::cout << "Terminating!" << std::endl;
        exit(1);

    }

};

int main()

{

    myterm ti;
    evtrec er;

    std::cout << "Hello c++ world" << std::endl;
    do { ti.event(&er); } while (1);

}
```

The event procedures are a way to break the rigid formalism of event loop design. In the event loop model, the event loop must be changed anytime there is new code that needs to handle events. Using event procedures, new handlers can be added without such modification. This enhances modularity, since the new code may be in a different module. This aids the extendibility of the system.

Event procedures are also a way to obfuscate a program by making it less obvious where the flow of control is in the program. The advantage to the event loop model is that it creates clear flow of control, even when handling asynchronous user generated events.

8.18 Timers

Timers allow a **terminal** program to perform periodic events, such as screen updates, and keeping track of time. From 1 to 10 timers are available, numbered 1..10. Each timer is given a time to measure, in 100 Microsecond counts (see 7.3 “Time and Date” in **services** for more details). When the timer is done, it sends an event to the event queue.

Timers can either simply stop when their time is done, or they can automatically start timing their original set time again. This is the "recurrent" mode, and it's useful when an activity needs to be performed periodically for the life of the program. For example, updating the screen on an active program, or checking when to update a clock display.

Timers are set by **pa_timer(f,i,t,r)**. A timer can be stopped or "killed" by **pa_killtimer()**. Killing a timer that is not active will not generate an error. This allows a single run timer to be killed without timing out during the call.

Due to the queuing nature of the event system, there is no guarantee about the accuracy of a timer. It can arrive later than its set time. If the program is late getting back to the event queue, or is busy with other events that occur before the timer, receipt of the timer event can be very late. All that receiving a timer event does is indicate that the time it measured has passed.

An example of timer use is the display of a clock, using the **services** time call. If a recurrent timer is set to go off on every second, the program should ***not*** simply advance the second on each timer event. Instead, the timer event should tell the program to read time to determine if the second has changed, and what value it currently has.

Implementations are free to limit the number of timers. Users should assume no more than 10 timers are available.

8.19 The Frame Timer

When performing animation, it is common to flip between screen buffers with select. The ideal time to perform a buffer flip is during the retrace time for the display, or the time it takes the display drawing hardware to reset to the top of the screen, and start drawing from the top again. Hiding the buffer flip in the retrace time can be key to making animations appear smooth.

The frame timer is a timer much like the standard timers, except that it is set automatically by the implementation. It is simply enabled or disabled, and gives an **pa_etframe** event when it times out. On hardware that is capable, the **pa_frametimer(f,e)** event is triggered by the beginning of the retrace cycle.

If an interrupt for the start of the retrace cycle is not available, then the frame timer is simply defaulted to a reasonable rate of redraw for animation, for example, 30 times per second.

8.20 Mouse

The mouse gives a position x,y on the screen, as well as from 1 to n buttons on it. A mouse actually gives its position as relative movements in x and y, but the system converts this to a screen position. The function **pa_mouse(f)** returns the number of mice attached to the system.

Mice generate two events. First, when the mouse moves, it generates position changes via the **pa_etmoumov** event. The program does not have to worry about where it is going. Each time the position changes, a new x, y position is posted as an event.

The second event generated by a mouse is mouse button asserts and deasserts, **pa_etmouba** and **pa_etmoubd**. An "assert" means a press of the button, and a "deassert" is the release of the button. Note that instead of an on/off status check as polled by the program, the assert and deassert events give exact notice of when the button changes state, and what it is changing to.

When there is more than one mouse per system, the default behavior should be to treat them as separate, but equal controls on the screen from the same user. When a mouse moves or changes button state, it gets control of the program. This matches the common use of multiple pointing devices, where two devices are alternated by one user. For example, a trackball and a mouse may be just two different input methods from one user.

Alternately, a second mouse could be a remote mouse over a network. This "collaborative computing" model allows two users to look at the same document, with separate mice. Advanced implementations of multiple mice such as these should be selected by the user.

Mice are subject to “rate limiting”. This means that the number of events per second reported by the mouse can be limited to no more than what the human viewer can perceive. This prevents the number of mouse events from affecting program performance.

8.21 Joysticks

From 0 to 4 joysticks may be supported. Each joystick can have from 0 to 3 "axes" of directions of travel. In addition, each joystick can have 0 to 4 buttons. The function **pa_joystick(f)** returns the number of joysticks in the system. The function **pa_joyaxis(f,j)** gives the number of axes on a given joystick. The function **pa_joybutton(f,j)** gives the number of buttons on a given joystick.

The messages **pa_etjoyba** or joystick button assert, and **pa_etjoybd** or joystick button deassert, give events for the assertion and deassertion of the buttons on a joystick. When any axis on a joystick moves, it generates a **pa_etjoymov**, or joystick movement, event. This event gives the relative setting of each axis of the joystick.

Unlike a mouse, a joystick is entirely relative. Each axis is represented by an integer. If the axis is not implemented it always reads 0. If the axis is deflected left, up or in, depending on the type of axis, then it is negative. If it is right, down, or out, it is positive. The axis is determined in its amount of deflection. If it is in the middle, it is 0. If it is deflected to the maximum, it is **INT_MAX**, with the sign giving the direction.

By convention, the axes on a joystick are:

1. Left/right, or slider.
2. Up/down
3. In/out

Joysticks are "rate limited" devices. This means that no matter how fast the joystick is "slewed", it will only produce an event about every 10 milliseconds at maximum rate. The reason for this is that humans cannot generally perceive events like movement of a pointer across the screen at a faster rate than this, so there is no point in updating faster than that, and flooding the input event queue.

A typical USB joystick with several axis and several buttons.

8.22 Function Keys

The system may have function keys, which are keys whose function are determined by the program. The number of function keys implemented are found with **pa_funkey(f)**. Function key messages are sent by the message **pa_etfun**. Typically at least 10 function keys are implemented, but **terminal** may implement many more. The Petit-Ami standard method for handling these is to give up to 10 predefined functions to the user, then let the user program the equivalent function for the rest of the function keys.

8.23 Automatic “hold” Mode

terminal implements a feature to help with legacy programs designed to the ANSI C serial I/O model. When a program exits that is unaware of the terminal model, the terminal window can abruptly close. This means that any printout from the program is lost.

The automatic hold mode keeps the window open unless a **pa_etterm** event is received, until the user specifically closes the window. This mode is enabled by default in systems were it is valuable, i.e., windowing systems.

There are times when the automatic hold mode can be a problem. For example, if a user displayed menu features a “quit” option, it would be incorrect to hold that window after the user has already closed the program.

To solve this, the **pa_autohold(e)** procedure can be used to set automatic hold mode off or on. With automatic hold mode off, the window exits immediately when the program does. Note that **pa_autohold(e)** does not take a file or window parameter.

8.24 Direct Writes

terminal accepts all standard ANSI C output methods, **putchar(c)**, **printf(fm,...)**, **fprintf(f,fm,...)** and others. However, it can be faster to output characters to the console directly, bypassing the normal file protocol layers inherent in the system. The procedure **pa_wrtstr(f,s)** outputs a character string directly to the terminal without any interpretation of control characters. It cannot be used with **pa_auto(f,e)** on.

8.25 Printers

terminal can be used to operate a printer using a subset of the **terminal** functionality. To model a printer, **terminal** uses a one page buffer that can be written using normal **terminal** commands to write to this "virtual screen" contained within the buffer. When the form-feed character is output ('\f'), the contents of the buffer is output to the printer as a whole page, and then a new page can be written.

When **terminal** is connected to a printer file, the following conditions are true:

1. There is no input file associated with the printer, neither event or virtual event methods function. **pa_event(f,er)** gives an error. None of the input devices work, and timer, mouse, joystick, function keys, the frame timer and the **pa_autohold(e)** procedure all give exceptions.
2. The **pa_select(f,u,d)** call does not function, and gives an error.
3. The dimensions given by **pa_maxx(f)** and **pa_maxy(f)** reflect the size of the printed page.
4. The exact set of attributes will be dependent on the printer. **pa_blink(f,e)**, of course, is never available.

A printer device is viewed as accepting a series of pages to be printed. The last page should be followed by outputting a form-feed ('\n'), to insure that a partial page is not left in the page buffer. **terminal** may automatically add a form-feed if that is not the last operation to the printer.

Since printer device mode is a subset of **terminal** functionality, it is possible for any program designed to output to a printer to have its output viewed on a terminal screen. Each page will then be presented to the user in turn.

The utility of using **terminal** to perform printer output, vs. using direct output to a printer, is that each page can be fully and easily formatted before outputting it.

8.26 Remote display

terminal is compatible with the use of "remote display" of presentation text. In this mode, the output from **terminal** is encoded and sent over a communications channel. This is done for both input and output functionality, and thus remote display mode is implemented without restriction of functionality.

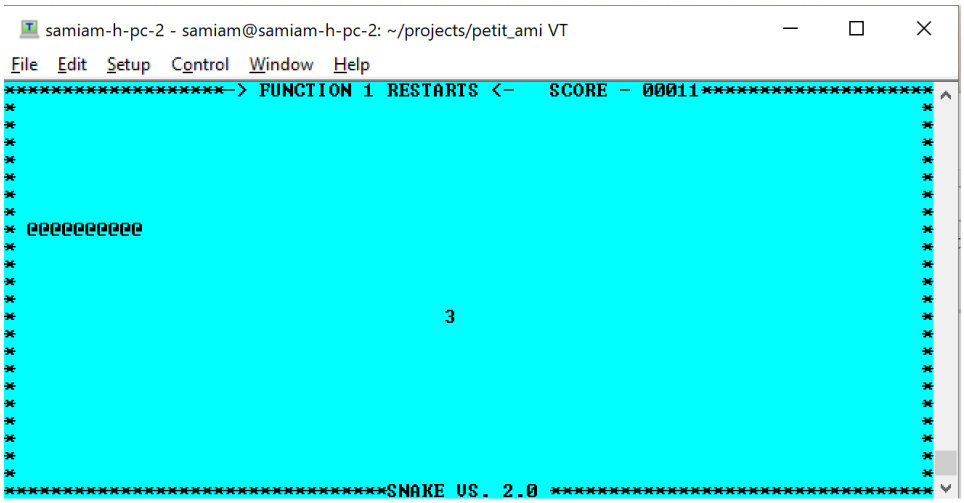
In order for remote display mode to work, the following conditions must be true:

1. All output functions must be applied in the order they were issued by the program.
2. All input events must be received in the order they were generated by the user.

Synchronization between output and input is done by the rule that all outstanding output must be completed before input is done. Thus any call to **pa_event(f,er)** causes a wait until all output operations are completed.

The exact format of the data passing over the communications channel to allow remote mode to function is system dependent.

Terminal is designed to operate efficiently with such remote displays. For example, there is no ability to read characters from the display.



An xterm ssh display is effectively a remote display. However, terminal may define it's own remote display protocol, or even the Xwindow protocol can be used. Of interest in this mode is that some features such as timers don't matter where they are implemented, server or client.

8.27 Terminal Objects In C++

All of the procedures, functions and other declarations in **terminal** are also available in a terminal object of the form:

```
/* object based interface */
class term {

FILE*      infile;
FILE*      outfile;

public:

/* constructor */
term();

/* methods */
void cursor(int x, int y);
int  maxx(void);
int  maxy(void);
void home(void);
void del(void);
void up(void);
void down(void);
```

```
void left(void);
void right(void);
void blink(int e);
void reverse(int e);
void underline(int e);
void superscript(int e);
void subscript(int e);
void italic(int e);
void bold(int e);
void strikeout(int e);
void standout(int e);
void fcolor(color c);
void bcolor(color c);
void autom(int e);
void curvis(int e);
void scroll(int x, int y);
int  curx(void);
int  cury(void);
int  curbnd(void);
void select(int u, int d);
void event(evtrec* er);
void timer(int i, int t, int r);
void killtimer(int i);
int  mouse(void);
int  mousebutton(int m);
int  joystick(void);
int  joybutton(int j);
int  joyaxis(int j);
void settab(int t);
void restab(int t);
void clrtab(void);
int  funkey(void);
void frametimer(int e);
void autohold(int e);
void wrtstr(char *s);
static void termCB(evtrec* er);

/* virtual callbacks */
virtual int evchar(char c);
virtual int evup(void);
virtual int evdown(void);
virtual int evleft(void);
virtual int evright(void);
virtual int evleftw(void);
virtual int evrightw(void);
virtual int evhome(void);
virtual int evhomes(void);
virtual int evhomel(void);
virtual int evend(void);
```

```
virtual int evends(void);
virtual int evendl(void);
virtual int evscrl(void);
virtual int evscrr(void);
virtual int evscru(void);
virtual int evscrd(void);
virtual int evpagd(void);
virtual int evpagu(void);
virtual int evtab(void);
virtual int eventer(void);
virtual int evinsert(void);
virtual int evinsertl(void);
virtual int evinsertt(void);
virtual int evdel(void);
virtual int evdell(void);
virtual int evdelcf(void);
virtual int evdelcb(void);
virtual int evcopy(void);
virtual int evcopyl(void);
virtual int evcan(void);
virtual int evstop(void);
virtual int evcont(void);
virtual int evprint(void);
virtual int evprintb(void);
virtual int evprints(void);
virtual int evfun(int k);
virtual int evmenu(void);
virtual int evmouba(int m, int b);
virtual int evmoubd(int m, int b);
virtual int evmoumov(int m, int x, int y);
virtual int evtim(int t);
virtual int evjoyba(int j, int b);
virtual int evjoybd(int j, int b);
virtual int evjoymov(int j, int x, int y, int z);
virtual int evresize(void);
virtual int evterm(void);

}; /* class term */
```

The description of each method in a **term** object appears with the same methods as the module terminal in 8.28.

A **term** object contains an event record, so it is not necessary to specify an external event record in the event procedure (nor possible).

A **term** object can be created as follows:

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <terminal.hpp>

using namespace terminal;

int main()

{

    term   ti; /* terminal object */
    evtrec er; /* event record */

    printf("hello c++ world\n");
    do { ti.event(&er);
        if (er.etype == etterm) exit(1);
    } while (er.etype != etenter);

}
```

As in the procedural interface to terminal, events in the **term** class can be registered as callbacks via the virtual procedures. However, just as in the procedural interface, such callbacks do not function unless the event method for the **term** object is called.

8.28 Procedures, functions and methods in terminal

For C++ optional parts of the specification of each function is denoted by []. The contents within the brackets are optional in the C++ interface. For all functions using an input file, the default for the file, if the parameter is not specified, is **stdin**. For all functions using an output file, the default for the file, if the parameter is not specified, is **stdout**.

void [pa_]cursor([FILE* f,] int x, int y);

Set cursor location for output surface file **f** in **x** and **y**.

int [pa_]maxx[(FILE* f)];

Find maximum screen location **x** in output surface file **f**.

int [pa_]maxy[(FILE* f)];

Find maximum screen location **y** in output surface file **f**.

void [pa_]home[(FILE* f)];

Send cursor to 1,1 location (upper left of screen) in output surface file **f**.

void [pa_]del[(FILE* f)];

Back up cursor by one character in output surface file **f**, and erase character at that location. If the cursor is at the left side of the screen, and automatic mode is on, the cursor will be moved up one line, and to the right of screen. If the cursor is at the top of the screen, extreme left, then the screen will be scrolled down one line, and the cursor moves to the right side of the screen. If automatic mode is off, the cursor simply moves left.

void [pa_]up[(FILE* f)];

Move cursor up one line in output surface file **f**. If the cursor is already at the top line, and automatic mode is on, the screen will be scrolled down, and the cursor remains at the same position. If automatic mode is off, the cursor simply moves up one line.

void [pa_]down[(FILE* f)];

Move cursor down one line in output surface file **f**. If the cursor is already at the bottom line, and automatic mode is on, the screen will be scrolled up, and the cursor remains at the same position. If automatic mode is off, the cursor simply moves down one line.

void [pa_]left[(FILE* f)];

Back up cursor by one character in output surface file **f**. If the cursor is at the left side of the screen, and automatic mode is on, the cursor will be moved up one line, and to the right of screen. If the cursor is at the top of the screen, left, then the screen will be scrolled down one line, and the cursor moves to the right side of the screen. If automatic mode is off, simply moves left one character.

void [pa_]right[(FILE* f)];

Move forward by one character in output surface file **f**. If the cursor is at the right side of the screen, and automatic mode is on, the cursor will be moved down one line, and to the left of screen. If the cursor is at the bottom of the screen, right, then the screen will be scrolled up one line, and the cursor moves to the left side of the screen. If automatic mode is off, simply moves right one character.

void [pa_]blink([FILE* f,] int e);

If **e** is 1, causes all further characters written to the screen to appear in blinking text in output surface file **f**. if **e** is 0, blink is turned off.

void [pa_]reverse([FILE* f,] int e);

If **e** is 1, causes all further characters written to the screen to appear in reverse text in output surface file **f**. If **e** is 0, reverse mode is turned off.

void [pa_]underline([FILE* f,] int e);

If **e** is 1, causes all further characters written to the screen to appear in underlined text in output surface file **f**. If **e** is 0, underline mode is turned off.

void [pa_]superscript([FILE* f,] int e);

If **e** is 1, causes all further characters written to the screen to appear in superscript text in output surface file **f**. If **e** is 0, superscript mode is turned off.

void [pa_]subscript([FILE* f,] int e);

If **e** is 1, causes all further characters written to the screen to appear in subscript text in output surface file **f**. If **e** is 0, subscript mode is turned off.

void [pa_]italic([FILE* f,] int e);

If **e** is 1, causes all further characters written to the screen to appear in italic text in output surface file **f**. If **e** is 0, italic mode is turned off.

void [pa_]bold([FILE* f,] int e);

If **e** is 1, causes all further characters written to the screen to appear in bold text in output surface file **f**. If **e** is 0, bold mode is turned off.

void [pa_]strikeout([FILE* f,] int e);

If **e** is 1, causes all further characters written to the screen to appear in strikeout text in output surface file **f**. If **e** is 0, strikeout mode is turned off.

void [pa_]standout([FILE* f,] int e);

If **e** is 1, causes all further characters written to the screen to appear in standout text in output surface file **f**. Standout is assigned to the first mode possible from the following order:

Reverse.
Underline.
Bold
Italic.
Strikeout.
Blink.

If none of those modes are available, **standout** is a no-op. If **e** is 0, standout mode is turned off.

void [pa_]fcolor([FILE* f,] pa_color c);

Sets the foreground, or text color, to the color **c** in output surface file **f**.

void [pa_]bcolor([FILE* f,] pa_color c);

Sets the background, or space color, to the color **c** in output surface file **f**.

void [pa_][m]auto([FILE* f,] int e);

Turns automatic mode on if **e** is 1, or off if **e** is 0, in output surface file **f**. Note that the mauto() name must be used in C++.

void [pa_]curvis([FILE* f,] int e);

Turns cursor visibility on if **e** is 1, or off if **e** is 0, in output surface file **f**.

void [pa_]scroll([FILE* f,] int x, int y);

Scroll in arbitrary directions. The update screen in surface file **f** is scrolled according to the differences in **x** and **y**. Uncovered areas on the screen appear in the current background color.

int [pa_]curx[(FILE* f)];

Find the current **x** location of the cursor in output surface file **f**.

int [pa_]cury[(FILE* f)];

Find the current **y** location of the cursor in output surface file **f**.

int [pa_]curbnd[(FILE* f)];

Check cursor in bounds. Returns true if the cursor is currently within the bounds of the screen in output surface file **f**.

void [pa_]select([FILE *f,] int u, int d);

Select buffer to update and display. Selects the active buffer for update **u**, and display **d** in output surface file **f**. The update buffer will receive the result of all writes. The display buffer will be shown on screen.

terminal provides at least 10 screen buffers, numbered 1 to n.

void [pa_]event([FILE* f,] pa_evtrec* er);

Get next event. Retrieves the next event from the input queue from the terminal input file **f** to event record **er**. If there is no event ready, the program will wait.

void [pa_]timer([FILE* f,] int i, int t, int r);

Set timer active in output surface file **f**. The timer **i** will be set to run for time **t**. If the repeat flag **r** is set, then the timer will automatically repeat when the time expires. If the timer is already in use, then it will cease its current timing, and perform the new time.

void [pa_]killtimer([FILE* f,] int i);

Stop timer in output surface file **f**. Stops the timer **i**. If the timer is not active, no error is reported.

int [pa_]mouse[(FILE *f)];

Returns the number of mice in output surface file **f**.

int [pa_]mousebutton([FILE* f,] int m);

Returns the number of buttons on a given mouse **m** in output surface file **f**.

int [pa_]joystick[(FILE* f)];

Returns the number of joysticks in the system in output surface file **f**.

int [pa_]joybutton([FILE* f,] int j);

Returns the number of buttons on the given joystick **j** in output surface file **f**.

int [pa_]joyaxis([FILE* f,] int j);

Returns the number of axes on the given joystick **j** in output surface file **f**.

void [pa_]settab([FILE* f,] int t);

Set new tab. Sets a new tab location at **t** in output surface file **f**.

void [pa_]restab([FILE* f,] int t);

Reset tab. Removes the tab at location **t** in output surface file **f**. If there is not a tab set there, it is not an error.

void [pa_]clrtab[(FILE* f)];

Clear all tabs. All tabs are removed from the tabbing table in output surface file **f**.

int [pa_]funkey[(FILE* f)];

Returns the number of function keys available in output surface file **f**.

void [pa_]frametimer([FILE* f,] int e);

Enables or disables the framing timer in output surface file **f**. If **e** is 1, the frame timer is enabled. If **e** is 0, it is disabled. The frame timer gives frame timer events, which occur approximately on each refresh of the display screen. It may be tied to the refresh hardware, or may be simulated via a timer.

void [pa_]autohold(int e);

Sets the state of automatic hold. If **e** is 1, **autohold** is enabled. If **e** is 0, **autohold** is disabled. **autohold** determines if the window will exit immediately if the program self terminates. If an exit was not ordered via the user interface, the display is held until it is.

void [pa_]wrtstr([FILE* f,] char *s);

Writes the string **s** directly to the output surface file **f**. No control character interpretation is done. This procedure is used to perform efficient writes to the display surface without per-character overhead.

It is an error to call this routine when **pa_auto()** is enabled.

void [pa_]eventover(pa_evtcod e, pa_pevthan eh, pa_pevthan* oeh);

Overrides the event **e** with the new handler function pointer **eh**, and returns the old event handler function pointer in **oeh**. The event handler function call is hooked, meaning that each new event function handler that overrides the original can chain to the last. If the event handler function does not care to process the event, it calls the old event handler to continue the chain. If no overrider wants to handle the event, it goes back to the caller of **pa_event().**

void [pa_]eventsover(pa_pevthan eh, pa_pevthan* oeh);

Overrides all events with the new handler function pointer **eh**, and returns the old event handler function pointer in **oeh**. The event handler function call is hooked, meaning that each new event function handler that overrides the original can chain to the last. If the event handler function does not care to process the event, it calls the old event handler to continue the chain. If no overrider wants to handle the event, it goes back to the caller of **pa_event().**

void [pa_]sizbuf([FILE* f,] int x, int y);

Sets the size of the buffer used to draw into. **x** and **y** indicate the width and height, respectively, of the buffer surface in window **f**.

8.29 Events in terminal

For each item, both the event structure section and the virtual function is presented. See the description of the event record (8.10 “Advanced Input” or 8.11 “Advanced Input in C++”) for the format of the entire record. For events, the parameters are fields of the structure. For overrides, they are formal parameters.

Event: [pa_]etchar
Override: int evchar(char echar);

Returns a keyboard character **echar**.

Event: [pa_]ettim
Override: int evtim(int timum);

Indicates the timer according to the timer handle **timnum** has expired.

Event: [pa_]etmoumov
Override: int evmoumov(int mmoun, int moupx, int moupy);

The mouse with handle **mmoun** has moved, to the position indicated by **moupx** and **ymoupy**.

Event: [pa_]etmouba
Override: int evmouba(int amoun, int amoub);

The mouse with handle **amoun** asserted the button **amoubn**.

Event: [pa_]etmoubd
Override: int evmoubd(int dmoun, int dmoubn);

The mouse with handle **dmoun** deasserted the button **dmoubn**.

Event: [pa_]etjoyba
Override: int evjoyba(int ajoyn, int ajoybn);

The joystick with handle **ajoyn** asserted the button **ajoybn**.

Event: [pa_]etjoybd
Override: int evjoybd(int djoyn, int djoybn);

The joystick with handle **djoyn** asserted the button **djoybn**.

Event: [pa_]etjoymov
Override: int evjoymov(int mjoyn, int joypx, int joypy, int joypz);

The joystick with handle **mjoyn** moved, and the coordinates **joypx**, **joypy** and **joypz**. The values of each axis are between –**INT_MAX**..**INT_MAX**. The number of axis actually present in the given joystick are given by the function **pa_joyaxis()**. The value returned by an unimplemented axis is undefined.

Event: [pa_]etfun
Override: int evfun(int fkey);

A function key was sent from the keyboard, with **fkey** giving the number of the key.

Event: [pa_]etup
Override: int evup(void);

The key for move cursor up was sent from the keyboard.

Event: [pa_]etdown
Override: int evdown(void);

The key for move cursor down was sent from the keyboard.

Event: [pa_]etleft
Override: int evleft(void);

The key for move cursor left was sent from the keyboard.

Event: [pa_]etright
Override: int evright(void);

The key for move cursor right was sent from the keyboard.

Event: [pa_]etleftw
Override: int evleftw(void);

The key for move cursor left word was sent from the keyboard. This indicates the cursor should be moved left one “word”, or over any series of non-space characters.

Event: [pa_]etrightw
Override: int evrightw(void);

The key for move cursor right word was sent from the keyboard. This indicates the cursor should be moved right one “word”, or over any series of non-space characters.

Event: [pa_]ethome
Override: int evhome(void);

The key for move cursor to the home position in the document (top extreme left) was sent from the keyboard.

Event: [pa_]ethomes
Override: int evhomes(void);

The key for move cursor to the home position in the screen (top extreme left) was sent from the keyboard.

Event: [pa_]ethomel
Override: int evhomel(void);

The key for move cursor to the home position in the line (extreme left) was sent from the keyboard.

Event: [pa_]etend
Override: int evend(void);

The key for move cursor to the end position in the document (bottom extreme right) was sent from the keyboard.

Event: [pa_]etends
Override: int evends(void);

The key for move cursor to the end position in the screen (bottom extreme right) was sent from the keyboard.

Event: [pa_]etendl
Override: int evendl(void);

The key for move cursor to the end position in the line (extreme right) was sent from the keyboard.

Event: [pa_]etscrl
Override: int evscrl(void);

The key for scroll screen left one character was sent from the keyboard.

Event: [pa_]etscrr
Override: int evscrr(void);

The key for scroll screen right one character was sent from the keyboard.

Event: [pa_]etscru
Override: int evscru(void);

The key for scroll screen up one character was sent from the keyboard.

Event: [pa_]etscrd
Override: int evscrd(void);

The key for scroll screen down one character was sent from the keyboard.

Event: [pa_]etpagd
Override: int evpagd(void);

The key for page down was sent from the keyboard.

Event: [pa_]etpagu
Override: int evpagu(void);

The key for page up was sent from the keyboard.

Event: [pa_]ettab
Override: int evtab(void);

The key for enter tab was sent from the keyboard.

Event: [pa_]etenter
Override: int eventer(void);

The key for enter line was sent from the keyboard.

Event: [pa_]etinsert
int evinsert(void);

The key for insert block was sent from the keyboard.

Event: [pa_]etinsertl
Override: int evinsertl(void);

The key for insert line was sent from the keyboard.

Event: [pa_]etinsertt
Override: int evinsertt(void);

The key for insert toggle was sent from the keyboard. The action should be to toggle the state of the insert/overwrite flag used to determine if new typed text overwrites previous text on screen, or inserts new text between existing characters.

Event: [pa_]etdel
Override: int evdel(void);

The key for delete block was sent from the keyboard.

Event: [pa_]etdell
Override: int evdell(void);

The key for delete line was sent from the keyboard.

Event: [pa_]etdelcf
Override: int evdelcf(void);

The key for delete character forward was sent from the keyboard. This indicates the character to the right of the cursor should be deleted.

Event: [pa_]etdelcb
Override: int evdelcb(void);

The key for delete character backward was sent from the keyboard. This indicates the character to the left of the cursor should be deleted.

Event: [pa_]etcopy
Override: int evcopy(void);

The key for copy block was sent from the keyboard. This indicates the currently selected block should be copied.

Event: [pa_]etcopyl
Override: int evcopyl(void);

The key for copy line was sent from the keyboard. This indicates the current line should be copied.

Event: [pa_]etcan
Override: int evcan(void);

The key for cancel current operation was sent from the keyboard. This indicates the operation in progress should be canceled.

Event: [pa_]etstop
Override: int evstop(void);

The key for stop current operation was sent from the keyboard. This indicates the operation in progress should be stopped.

Event: [pa_]etcont
Override: int evcont(void);

The key for continue current operation was sent from the keyboard. This indicates the operation in progress should be continued if stopped previously.

Event: [pa_]etprint
Override: int evprint(void);

The key for print current document was sent from the keyboard. This indicates the current document should be printed in entirety.

Event: [pa_]etprintb
Override: int evprintb(void);

The key for print current block was sent from the keyboard. This indicates the current selected bock, if it exists, should be printed.

Event: [pa_]etprints
Override: int evprints(void);

The key for print current screen was sent from the keyboard. This indicates the current screen should be printed.

Event: [pa_]etmenu
Override: int evmenu(void);

The key for display menu was sent from the keyboard. This indicates the menu, if any, should be displayed.

Event: [pa_]etresize
Override: int evmenu(int rszx, int rszy);

The window for terminal has resized. The new width is **rszx**, and the new height is **rszy**. The **etresize** event will only be sent if terminal is in a windowed system. Since terminal is a buffered model, it can be ignored, since the drawing surface, and the geometry parameters **maxx()** and **maxy()** will not be affected. If the client chooses to resize to follow the window resize, **sizbuf()** is used to change the geometry.

Event: [pa_]etterm
Override: int evresize(void);

The key for terminate program was sent from the keyboard. This indicates the program should be exited.

If this event is received, then the user ordered the exit. This means that automatic hold mode, if enabled, will be bypassed, and the program closed immediately.

9 Graphical Interface Library

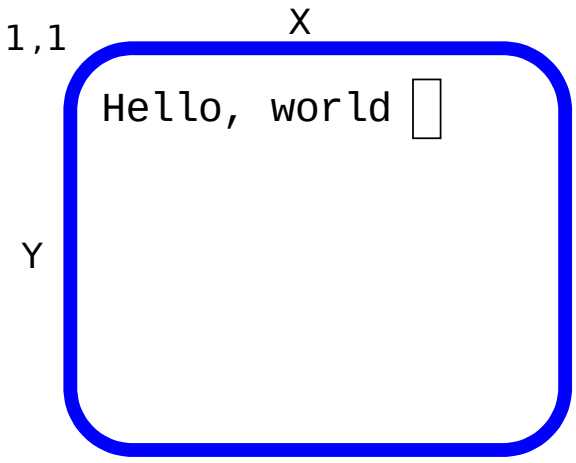
The graphical library **graphics** extends the **terminal** model by adding graphical output procedures.

Since it is completely upward compatible with **terminal**, and standard ANSI C serial output modes, any program from ANSI C, or **terminal** compliant ansi c will run under **graphics**. In the most advanced modes of **graphics**, ordinary ANSI C I/O statements can still be used to output text, so all of the input and output formatting functions of ANSI C still work.

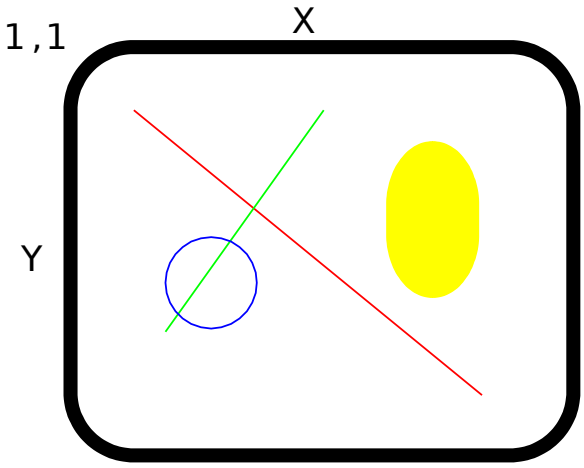
graphics will handle any graphics task. Besides drawing features, it supports double page animation. Combined with the sound library **sound**, full graphical games are possible.

9.1 Terminal model

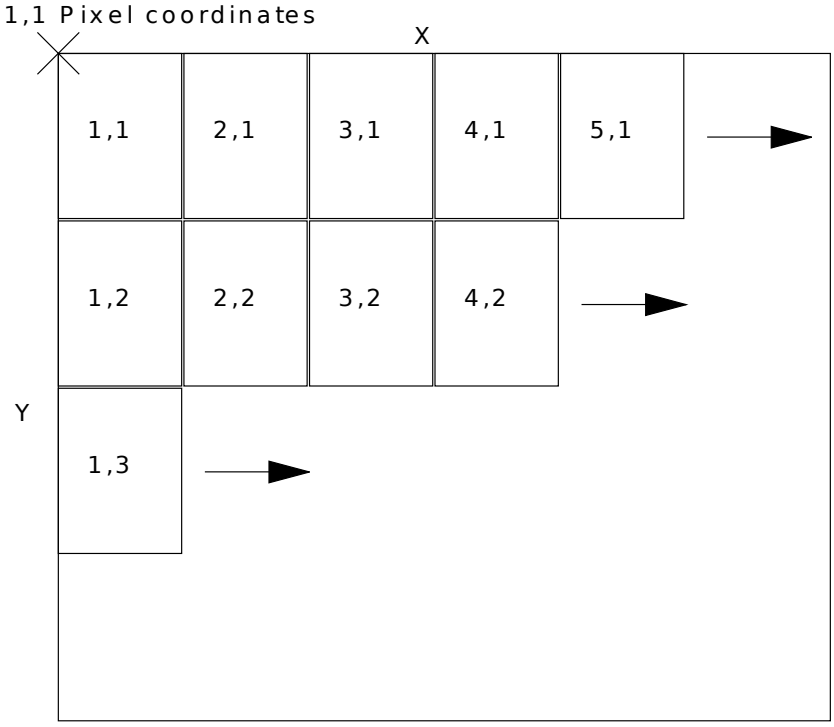
graphics emulates **terminal** by setting up a "character grid" across the pixel based screen. Each "cell" on the grid matches the pixel height and width of a character. The character font is set to a fixed font by default when **graphics** starts. This gives every character in the font the same height and width. The text drawing mode is set such that both the foreground (the inside of the characters) and the background (the space behind the characters) are drawn, overwriting any content of each character cell as it is written.



Is implemented in a graphics coordinate screen:



Each increment is called a picture element, or pixel, and represents the smallest drawable dot on the screen. Each character of the terminal mode is mapped onto a square area of the drawing surface as follows:



This mode can be kept, while drawing other figures on the same text surface. Alternately, the cursor can be set to any arbitrary pixel on the screen, and text written anywhere, down to the pixel. Also, the font can be changed to a proportional one for a more pleasing look. Because automatic mode relies on

characters being neatly placed on the grid, it must be turned off before the cursor leaves the grid or becomes a proportional font.

When **pa_auto()** is turned off, the grid is still useful. Although the character spacing varies, the line spacing is still valid. This means that the grid is no longer useful in the x direction, but it is still useful in the y direction. In addition, the origin in x is still valid. The result is that a series of lines printed with end-of-lines will do the right thing with **pa_auto()** off, namely present a series of left justified lines at the x origin (flush left) with the correct spacing.

9.2 Graphics Coordinates

The total size of the graphics screen is found by **pa_maxxg(f)** and **pa_maxyg(f)**, which return the maximum pixel index in x and y. The pixel coordinates on the screen are from 1,1 to **pa_maxxg(f)**, **pa_maxyg(f)**. The cursor can be set to any pixel position by **pa_cursorg(f, x, y)**. The current location of the cursor in pixel terms is found by **pa_curxg(f)** and **pa_curyg(f)**.

9.3 Character Drawing

There can be any number of fonts available on the system, including both fixed space fonts, and proportional fonts that vary in the width of characters. The number of fonts in the system can be found with the **pa_fonts(f)** function. Fonts are chosen by logical number with **pa_font(f, c)**, where **c** is the font code, 1 to **pa_fonts(f)**. There are two methods to determine what font is assigned to a particular font code. The first is the standard font codes, the second is the font name system. Standard fonts are numbers for commonly used fonts in the system. These are commonly available fonts the program may need.

The standard font codes are:

Terminal Font: **PA_FONT_TERM: 1**

This is the default font set up by **graphics** when it starts. It's a fixed font. It also cannot be superscripted, subscripted, bold or italic, because these modes change the size of the font.

Book Font: **PA_FONT_BOOK: 2**

This is a serif font, and is good for general purpose text such as what a paragraph in a book is written in. This is the most common proportional font.

Sign Font: **PA_FONT_SIGN: 3**

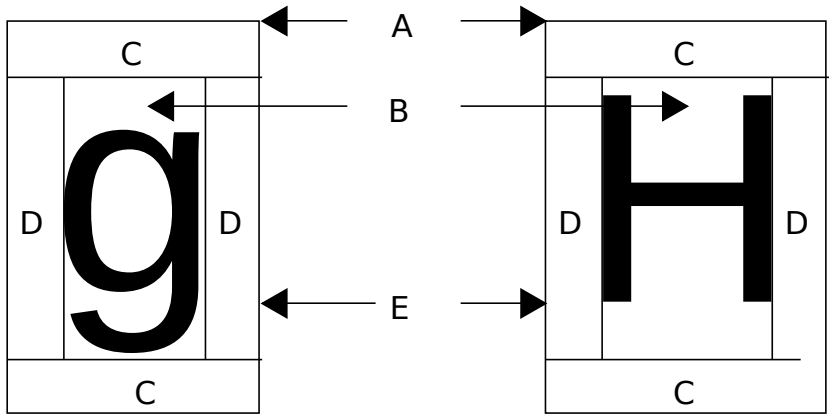
This is a no serif font (sans serif), and is best for headings, titles and similar uses, as in road signs and other signs. It's a proportional font.

Technical Font: **PA_FONT_TECH: 4**

The technical font is a fixed font that is guaranteed to be able to scale to any arbitrary size. This font is used to label drawings and engineering documents. Before the technology existed to create arbitrary fonts in any point size, the technical font was done with stroked vector graphics. Now, it is more likely to be equivalent to the sign font.

The first four fonts corresponding to the standard fonts are always present, so **pa_fonts()** will always be 4 or greater.

The name of an installed font can be found by **pa_fontnam(f, fc, fns)**, where **fc** is the font code, 1 to **pa_fonts(f)**, and **fns** returns the descriptive string for the font, such as "Helvetica". The application should use the standard fonts by default, then present the system fonts, by name to the user, and let the user choose one of them.



The Parts of the box are:

Label	Part
A	Character box
B	Character bounding box
C	Line space
D	Character space
E	Baseline

The size of each character is set by **pa_fontsiz(f, n)**, where n is the height of the font in pixels. The reason character sizes are set by their height is because proportional fonts vary in the width of the character. The height never varies. The current height of the font is found by **pa_chrsizy(f)**. Its width is found with **pa_chrsizx(f)**. When a proportional font is active, **pa_chrsizx(f)** returns the width of a space in that font, which is always as wide or wider than the widest character in that font.

Besides the basic font, extra space can be added between lines (known as "leading" in typography, for the lead strips used between type lines) with **pa_chrspcy(f, n)**. n is the number of pixels of extra space to add between lines. Extra space between characters is added with **pa_chrspcx(f, n)**, where n is the number of pixels of extra space to add between characters.

The graphical cursor is placed at the upper left of the character box for the next character to be drawn. If it is needed to know exactly where the character will rest if drawn on a line. The offset from the top of the character box to the baseline of the text is found with the function **pa_baseline(f)**.

In typography, fonts and characters are measured by points, of which there are 28.35 points per centimeter. Point measurement implies that the exact size of objects drawn on the screen is known. This can be determined by the functions **pa_dpmx(f)** and **pa_dpmy(f)**, which return the "dots per meter" or pixels in one meter for both x and y. The reason it can be two different measures for the two different axes is that the display may not have square pixels or a 1:1 aspect ratio.

To find a given point size in terms of the height needed for the character, it is found by:

```
pa_dpmy(f)/2835*point size
```

The screen aspect ratio can also be found from these calls, it is:

```
pa_dpmx(f)/pa_dpmy(f)
```

9.4 String Sizes and Kerning

When writing text to the screen in proportional font, it is often needed to know exactly how much space the string will take up on the screen, down to the pixel. This amount of space can change not only because of the variable width of proportional fonts, but also because of an effect called "kerning". When a string of characters is draw together, the system can apply "kerning" to characters in the string. Kerning means to fit the individual letters together, like puzzle pieces, to come up with a tighter spacing

than is normally possible. For example, "A" and "V" together as "AV" can typically be kerned together to take less space because they can overlap. To find the exact number of pixels in x that a string will occupy, the function **pa_strsiz(f, s)** is used, where s is the string. The size of the string in y does not change, and can be found with **pa_chrsizy**. To find the exact position, in pixels offset from the beginning of the string in x, of a given character, use **pa_chrpos(f, s, n)**, where **s** is the string, and **n** is the index of the character to find the x offset of, from 0 to n.

9.5 Justification

Justification is the spreading of spacing through a string of characters to fit a given space. If the string will fit into the space is found with **pa_strsiz(f, s)**, and checking if the resulting pixels required are less than or equal to the space they will occupy as justified. The character string is written in justified mode with **pa_writejust(f, s, n)**, where **s** is the string to write, and **n** is the number of pixels to fit it in. If the number of pixels allowed for is not enough, the string will be larger than the requested number. The offset, in x, of a given justified character, is found with **pa_justpos(f, s, p, n)**, where **s** is the string, **p** is the offset of the character you are interested in, and **n** is the total number of pixels to fit the string in, as in **pa_writejust()**.

9.6 Effects

graphics expands the effects in **terminal**. For smaller character baselines, **pa_condensed(f, b**), is used. For larger character baselines, **pa_extended(f, b)** is used**.**

In addition to normal **pa_bold(f,b)**, there are also **pa_light(f,b)**, **pa_xlight(f, b)**, and **pa_xbold(f,b)** effects. For lighter than normal, extra light, and extra bold modes.

Characters will have an embossed look with **pa_hollow(f, b)** and **pa_raised(f, b)**. Hollow makes the character look sunken, and raised makes it look as if coming off the page.

9.7 Tabs

graphics extends the character level of tabbing in **terminal** with procedures that can set tabs on an individual pixel. **pa_settabg(f, x)** sets a tab at the pixel **x**. **pa_restabg(f, x)** resets the tab at pixel **x**. The **terminal** procedure **clrtab(f)** clears all tabs, including pixel level tabs.

9.8 Colors

The simple eight colors from **terminal** are still available, with the addition of two new calls that allow access to the full range of colors an advanced graphic system provides. **pa_fcolorg(f, r, g, b)** sets the foreground color from values of red **r**, green **g**, and blue **b**. Similarly, **pa_bcolorg(f, r, g, b)** sets the background color from rgb values. The values of the colors are ratioed. This means that instead of an absolute number, the possible colors are ratioed from 0 to **INT_MAX**, where 0 is dark, and **INT_MAX** is saturated color. Color ratios allow the true color range implemented by the system to be hidden.

9.9 Drawing Modes

When colors, background or foreground, are drawn on the screen, they can be in a number of mixing modes. Mixing modes govern how the new color is laid over the old. The modes are mutually exclusive, so the setting of a new mix mode for a given color deactivates the old mode. If the old color is simply to be overwritten, then **pa_fover(f)** or **pa_bover(f)** is appropriate. If the new color is to be xor'ed with the old color, then **pa_xor(f)** or **pa_bxor(f)** is used. If the new color is to be ignored, leaving the old

color underneath intact, use **pa_finvis(f)** or **pa_invis(f).** There is also **pa_fand(f), pa_band(f), pa_for(f)** and **pa_bor(f).**

There might not seem to be a use for an "invisible" color, but there are actually several uses. First, if the background is set invisible, the text can be overlaid on another pattern. In fact, this is the most common drawing mode, and most programmers will prefer to turn the background off and lay the backgrounds themselves. Similarly, leaving the background on, then setting the foreground invisible can be used to "stencil" letters with arbitrary patterns inside the letters.

pa_xor(f) mode is good for several things. First, if a series of figures, say rectangles, are to be laid, but the intersections between them are to be left visible, **pa_xor()** is the right mode. Second, **pa_xor()** can be used to place, and then remove a pattern, even a complex one, easily. This is used to allow the user to place figures by dragging them with the mouse to a new location. It can be used to draw "rubber band" boxes around selections.

pa_xor() mode has two rules of interest:

1. Any two colors, even the same colors, xored together, will give a third color, with the exception of black.
2. Having xored a drawing into the viewplane, xoring the same color and drawing into the viewplane again will restore the old drawing.

pa_xor() can be used for several special effects. However, **pa_xor()** does not tolerate inaccuracy. Xoring something back off the screen has to be done the same way it was put on, with the same parameters. Also, the mode of drawing in **graphics** is not compatible with some uses of the **pa_xor()** mode. Drawing a rectangle, for example, will result in "corner errors", because the rectangles are built from lines, and lines start and end at the coordinates of the box corners. Because one line starts at the same point another ends, they will xor mix together. The result is a recognizable point of off-color on each corner.

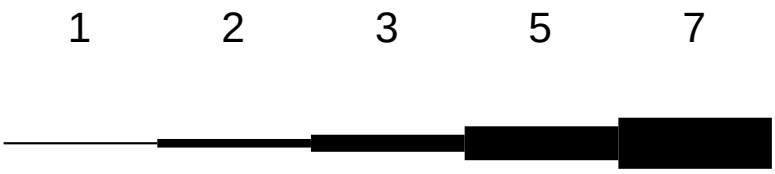
The best way to use xor is to xor a single figure onto the screen, then xor it back off. If complex figures are to be xored on and off the screen, xor is run backwards, that is, from the last figure drawn back to the first.

pa_and() and **pa_or()** modes are used to create stencils and other effects. For example, a drawing and'ed with a black stencil will remain black in the stencil area, and intact elsewhere.

9.10 Drawing Graphics

A graphics element that is not a character is referred to as a "figure". What **graphics** tries to do is provide a small toolset, that does not include figures that you could reasonably construct from the lower level figures. For example, there is no circle figure in **graphics**, because that is simply a special case of **ellipse**.

A parameter that applies to almost all figures is the width of lines. The width of a line usually defaults to 1, but may be more if a single line is unusable on the current display. This can easily happen on a very high resolution display. The line width can be set by the procedure **pa_linewidth(f, n)**, where **n** is the number of pixels for the line to use. There is no limit on the width of a line, and in fact, lines are a defacto way to draw arbitrary angle filled rectangles.



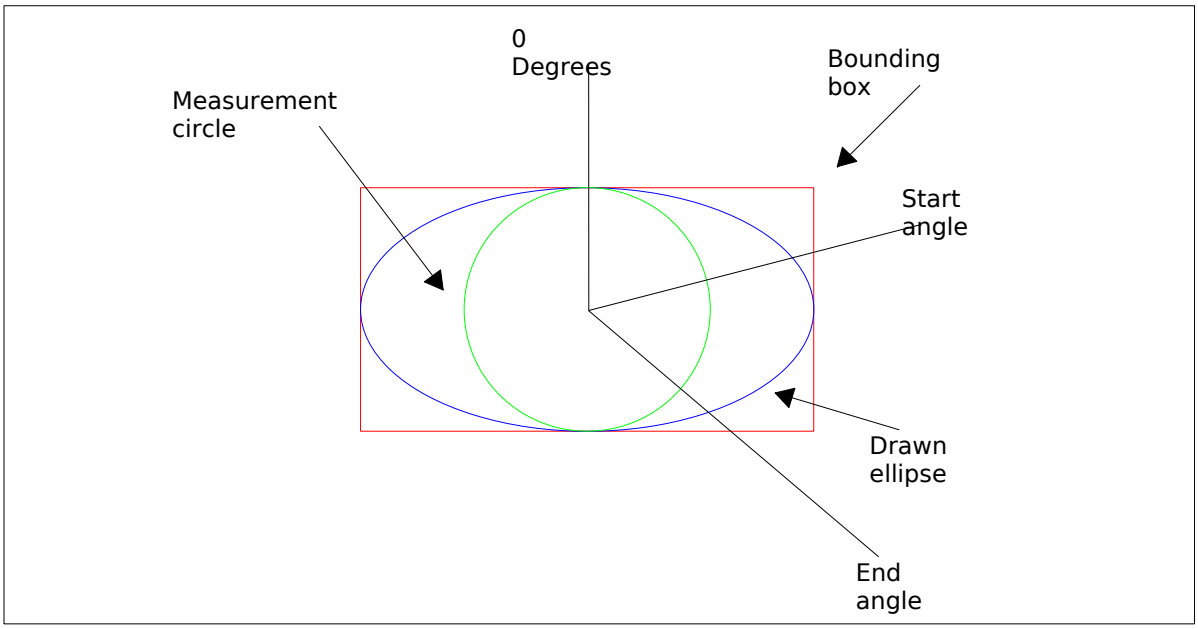
When the line width is set to an even number of pixels, an effect called "even line uncertainty" exists. If you draw a line between two points, and the line width is say, 2, one of the 2 pixels is going to be on the line, and the other could be to either side of it. It literally depends on how the math happens to round off.

To prevent this, line width should always be set to an odd number.

9.11 Figures

The fundamental figure in graphics is the line. A line is drawn, in the current **pa_linewidth()**, by **pa_line(f, x1, y1, x2, y2)**. A rectangle is drawn with **pa_rect(f, x1, y1, x2, y2)**, whose borders have the current **pa_linewidth()**. A filled rectangle is drawn with **pa_frect(f, x1, y1, x2, y2)**, whose interior is the foreground color. An ellipse is drawn with **pa_ellipse(f, x1, y1, x2, y2)**. The x and y parameters define a rectangle that contains the figure. The procedure **pa_fellipse(f, x1, y1, x2, y2)** draws a filled ellipse.

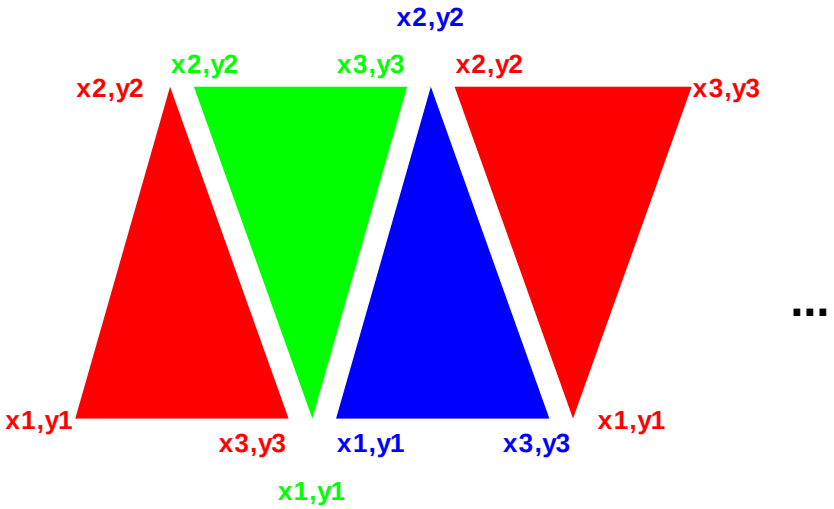
The procedure **pa_arc(f, x1, y1, x2, y2, rs, re)** draws an arc line around the ellipse formed by the rectangle formed by the x and y parameters. The start and end points of the arc are described by a special ratio notation that gives the angle. The angles in a 360 degree circle are described by a number from 0 to **INT_MAX**. 0 is the 0 degree, or top center, of the ellipse. The angles around the circle clockwise then go from 0 to **INT_MAX**, at which time a full 360 degrees have been traversed. For example, **INT_MAX** div 2 is 180 degrees, **INT_MAX** div 4 is 90 degrees, etc. The parameter **rs** gives the angle where the arc starts. The parameter **re** gives the angle where the arc ends. Arcs can be specified to cross **INT_MAX** back to zero, or use negative degrees, or any combination.



The procedure **pa_farc(f, x1, y1, x2, y2, rs, re)** draws a filled arc. The procedure **pa_fcord(f, x1, y1, x2, y2, rs, re)** draws a filled cord (a line bisecting the circle).

Rectangles with rounded corners can be drawn with **pa_rrect(f, x1, y1, x2, y2, xw, yw)**. The x and y parameters describe the bounding box, The **xw** and **yw** parameters describe the size of the ellipses that are placed in the corners to round the edges of the box. To draw a filled rounded rectangle, use **pa_frrect(f, x1, y1, x2, y2, xw, yw)**.

The general purpose shape **pa_ftriangle(f, x1, y2, x2, y2, x3, y3)** draws a filled triangle. Parting with convention, **graphics** does not give complex polygon procedures. Rather, you can build up polygons from triangles, and in any case, a high speed drawing engine in hardware would accept triangles only, so the lower level software would have to break up the polygon for you.



Example of polygon filled shape draw:

```
typedef struct { int x, y; } point;

void poly(FILE* f, point points[] /* ends with 0,0*/)

{

    int p = 0;
    pa_color c = pa_red;

    /* count points */
    while (points[p].x || points[p].y) p++;
    if (p < 3) return; /* minimum 3 points */
    p = 1; /* index 2nd point */
    while ((points[p].x || points[p].y) &&
           (points[p+1].x || points[p+1].y)) {

        /* connect origin, next two points */
        pa_ftriangle(f, points[0].x, points[0].y,
                        points[p].x, points[p].y,
                        points[p+1].x, points[p+1].y);
        p++;

    }

}
```

Single pixels can be set with **pa_setpixel(f, x, y)**.

9.12 Predefined Pictures

A picture, or a bitmap, is defined outside the program by a drawing application. Its format is typically operating system specific. **graphics** considers pictures to be a cached resource. A picture is loaded from a file by **pa_loadpict(f, p, fs)**. The string **fs** indicates the file name for the picture file. **p** is a logical picture number, from 1 to n, and indicates how you want to refer to the picture while its loaded into memory.

A logical picture is drawn onto the screen with **pa_picture(f, p, x1, y1, x2, y2)**. The parameter **p** indicates the logical picture to draw. The x and y parameters indicate the box that the picture is to be drawn into. **graphics** will scale or stretch the picture as needed to make the picture fit into the space given.

In order to determine the parameters of a picture, such as native size and aspect, the functions **pa_pictsizx(f, p)** and **pa_pictsizy(f, p)** are used. These give the native size of the picture in x and y, and the aspect ratio of the picture is then found with **pa_pictsizex(f, p)**/**pa_pictsizey(f, p)**.

When the picture is no longer needed, **pa_delpict(f, p)** removes it from cache.

9.13 Scrolling

As in **terminal**, **graphics** can scroll in arbitrary directions. It can also scroll down to the pixel, using the call **pa_scrollg(f, x, y)**. The parameters work the same way as the character position parameters of scroll, except that pixels are specified instead of characters.

9.14 Clipping

Clipping is automatic in **graphics**. Any figure drawn is clipped to the edges of the screen. If a figure is drawn entirely outside the screen bounds, it is clipped out.

9.15 Mouse Graphical Position

A new event, **pa_etmoumovg**, exists that gives mouse movements in pixels, not just characters. The old **pa_etmoumov** still occurs, and carries the character grid message. The **pa_etmoumovg** message happens when the mouse moves a pixel, and the **pa_etmoumov** message happens when the mouse moves a whole character cell. If you don't need the **pa_etmoumov[g]** message, you simply ignore it.

9.16 Animation

In **terminal**, the **select()** call was introduced, that switches between multiple screen buffers. This call is tailor made for double buffer animation, it works in **graphics** the same way. Double buffer animation works by having two screen buffers. One of them is drawn, and the other one is displayed. When a "frame" is drawn, i.e., the picture in the buffer currently being draw is complete, the drawing and display buffers are swapped, and then drawing begins on what used to be the display buffer, clearing it if necessary.

Double buffering removes any of the ongoing drawing operations from the active screen. Although computers do this quite fast, seeing drawing operations in progress tends to produce annoying flashing effects, sparkles and other effects during the drawing. In addition, although the worst interactions with

the painting of the graphics card memory to the display screen have disappeared, there can still be odd effects from the fact that the display is being refreshed across the image being drawn.

graphics supports more than double buffering (triple, quad or better). However, the advantages of this typically diminish as the amount of data being managed grows without a compensating gain in drawing speeds.

To reduce flicker effects the buffer is flipped when the display enters its retrace period. Syncing with the display eliminates the effects that occur when writing to the display while the display is drawing across it. The retrace period is when the refresh cycle has finished the last lines on the screen, and will head back to the top of the screen. This gives a short time while the display is not doing anything, so if the buffer swap can occur within the retrace, no effects will appear on the screen at all.

The solution is the **pa_etframe** message. **pa_etframe** is sent when the display enters refresh. If the system does not allow notification for retrace, the **pa_etframe** message is generated by a timer that keeps either the screen refresh rate, or 30 cycles per second if that cannot be determined.

9.17 Copy between buffers

Blocks of pixels can be copied between buffers with procedure **pa_blockcopyg(f, s, d, sx1, sy1, sx2, sy2, dx1, dy1, dx2, dy2)**. This copies a pixel block from a source bounding box to a destination box in the same or a different buffer. It is capable of both resizing the block, as well as using the write mode to place the pixels.

Copying between buffers is a very powerful technique to speed animation. Pictures can be constructed and processed in a non-displayed buffer, then copied quickly to a target buffer for display. A program with a lot of animation will typically have an offline buffer just to keep sprites and other drawn objects ready to present. The drawing mode can be used to create stencils and other drawing tools.

9.18 Buffer Follow mode

As in the **terminal** mode, the buffer follow mode can be used. When the program receives a **pa_etresize**, in addition to the the **rszx** and **rszy** size parameters, it receives the **rszxg** and **rszyg** size parameters. These can be used to set the buffer size in graphical terms with **pa_sizbufg(f,x,y).** Then the functions **pa_maxx(f)** and **pa_maxy(f)** will return the character size, and the functions **pa_maxxg(f)** and **pa_maxyg(f)** will return the size in graphics terms. As in **terminal**, this effectively destroys the existing buffers and creates new ones that are cleared to the background color.

9.19 Printers

As with **terminal**, **graphics** can output to a printer if the printer is graphics capable. The one page buffer contains sufficient pixels to allow a complete page with graphics to be rendered in the buffer, then output to the printer with by outputting a ‘\f’ (form-feed) operation.

See **terminal** for a list of restrictions on printer operation.

9.20 Remote display

Graphics is capable of using a remote display as in **terminal**. The same comments as in **terminal** apply to **graphics**.

9.21 Declarations

The declarations for **graphics** are very similar to **terminal**, with the addition of the mouse move graphical event, and the standard font codes.

```
#define PA_MAXTIM 10 /**< maximum number of timers available */

/* events */
typedef enum {
    pa_etchar,      /* ANSI character returned */
    pa_etup,        /* cursor up one line */
    pa_etdown,      /* down one line */
    pa_etleft,      /* left one character */
    pa_etright,     /* right one character */
    pa_etleftw,     /* left one word */
    pa_etrightw,    /* right one word */
    pa_ethome,      /* home of document */
    pa_ethomes,     /* home of screen */
    pa_ethomel,     /* home of line */
    pa_etend,       /* end of document */
    pa_etends,      /* end of screen */
    pa_etendl,      /* end of line */
    pa_etscrl,      /* scroll left one character */
    pa_etscrr,      /* scroll right one character */
    pa_etscru,      /* scroll up one line */
    pa_etscrd,      /* scroll down one line */
    pa_etpagd,      /* page down */
    pa_etpagu,      /* page up */
    pa_ettab,       /* tab */
    pa_etenter,     /* enter line */
    pa_etinsert,    /* insert block */
    pa_etinsertl,   /* insert line */
    pa_etinsertt,   /* insert toggle */
    pa_etdel,       /* delete block */
    pa_etdell,      /* delete line */
    pa_etdelcf,     /* delete character forward */
    pa_etdelcb,     /* delete character backward */
    pa_etcopy,      /* copy block */
    pa_etcopyl,     /* copy line */
    pa_etcan,       /* cancel current operation */
    pa_etstop,      /* stop current operation */
    pa_etcont,      /* continue current operation */
    pa_etprint,     /* print document */
    pa_etprintb,    /* print block */
    pa_etprints,    /* print screen */
    pa_etfun,       /* function key */
    pa_etmenu,      /* display menu */
    pa_etmouba,     /* mouse button assertion */
    pa_etmoubd,     /* mouse button deassertion */
    pa_etmoumov,    /* mouse move */
    pa_ettim,       /* timer matures */
    pa_etjoyba,     /* joystick button assertion */
    pa_etjoybd,     /* joystick button deassertion */
```

```
    pa_etjoymov,    /* joystick move */
    pa_etresize,    /* window was resized */
    pa_etterm,      /* terminate program */
    pa_etframe,     /* frame sync */
    pa_etmoumovg    /* mouse move graphical */

} pa_evtcod;
/* event record */
typedef struct {

    /* identifier of window for event */ int winid;
    /* event type */                      pa_evtcod etype;
    /* event was handled */               int handled;
    union {

        /* these events require parameter data */

        /** etchar: ANSI character returned */  char echar;
        /** ettim: timer handle that matured */ int timnum;
        /** etmoumov: */
        struct {

            /** mouse number */   int mmoun;
            /** mouse movement */ int moupx, moupy;

        };
        /* etmouba */
        struct {

            /** mouse handle */  int amoun;
            /** button number */ int amoubn;

        };
        /* etmoubd */
        struct {

            /** mouse handle */  int dmoun;
            /** button number */ int dmoubn;

        };
        /* pa_etjoyba */
        struct {

            /** joystick number */ int ajoyn;
            /** button number */   int ajoybn;

        };
```

```
        /* pa_etjoybd */
        struct {

            /** joystick number */ int djoyn;
            /** button number */   int djoybn;

        };
        /* pa_etjoymov */
        struct {

            /** joystick number */      int mjoyn;
            /** joystick coordinates */ int joypx, joypy, joypz;

        };
        /** pa_etresize */
        Struct {

            Int rszx, rszy, rszxg, rszyg;

        }
        /** pa_etfun */
        /** function key */ int fkey;
        /** etmoumovg: */
        struct {

            /** mouse number */   int mmoung;
            /** mouse movement */ int moupxg, moupyg;

        };

    };

} pa_evtrec, *pa_evtptr;
```

9.22 Functions in graphics

See **terminal** for the basic text functions. These are all implemented in **graphics**.

For all of the following functions, If the screen file **f** is not present, the default is the standard **stdout** or standard **stdin** file.

int pa_maxxg(FILE* f);

Returns the maximum pixel index x in output surface file **f**.

int pa_maxyg(FILE* f);

Returns the maximum pixel index y in output surface file **f.**

int pa_curxg(FILE* f);

Returns the current location of the cursor, in pixel units, for x in output surface file **f**.

int pa_curyg(FILE* f);

Returns the current location of the cursor, in pixel units, for y in output surface file **f**.

void pa_line(FILE* f, int x1, int y1, int x2, int y2);

Draw line. Draws a line between the point **x1**,**y1** to **x2**,**y2**, using the current line width, color and mode in output surface file **f**.

void pa_linewidth(FILE* f, int w);

Set line width. Sets the line drawing width at **w** pixels wide in output surface file **f**. Use of an odd number of pixels is recommended.

void pa_rect(FILE* f, int x1, int y1, int x2, int y2);

Draw rectangle. Draws a rectangle whose opposite corners are **x1**,**y1** and **x2**,**y2**. Uses the current line width, color and mode in output surface file **f**.

void pa_frect(FILE* f, int x1, int y1, int x2, int y2);

Draw filled rectangle. Draws a solid rectangle, whose opposite corners are **x1**,**y1** and **x2**,**y2**. Uses the current color and mode in output surface file **f**.

void pa_rrect(FILE* f, int x1, int y1, int x2, int y2, int xs, int ys);

Draw rounded rectangle. Draws a rectangle with rounded corners. The opposite corners of the bounding box are specified as **x1**,**y1** to **x2**,**y2**. The ellipses that specify the rounded corners are **xs** and **ys**, which specify the width and height of the ellipse. Uses the current line width, color and mode in output surface file **f**..

void pa_frrect(FILE* f, int x1, int y1, int x2, int y2, int xs, int ys);

Draw filled rounded rectangle. Draws a rectangle with rounded corners. The opposite corners of the bounding box are specified as **x1**,**y1** to **x2**,**y2**. The ellipses that specify the rounded corners are **xs** and **ys**, which specify the width and height of the ellipse. Uses the current color and mode in output surface file **f**.

void pa_ellipse(FILE* f, int x1, int y1, int x2, int y2);

Draw an ellipse. Draws an ellipse bounded by a box whose opposite corners are **x1**,**y1** to **x2**,**y2**. Uses the current line width, color and mode in output surface file **f**.

void pa_fellipse(FILE* f, int x1, int y1, int x2, int y2);

Draw a filled ellipse. Draws a solid ellipse bounded by a box whose opposite corners are **x1**,**y1** to **x2**,**y2**. Uses the current color and mode in output surface file **f**.

void pa_arc(FILE* f, int x1, int y1, int x2, int y2, int sa, int ea);

Draw an arc. Draws an arc on an ellipse, whose bounding box is **x1**,**y1** to **x2**,**y2**. The arc starts on the ellipse from the angle **sa**, to the angle **ea**. The angles are given in 360 degree to **INT_MAX** ratio form. Uses the current color and mode in output surface file **f**.

void pa_farc(FILE* f, int x1, int y1, int x2, int y2, int sa, int ea);

Draw a filled arc. Draws a filled arc on an ellipse, whose bounding box is **x1**,**y1** to **x2**,**y2**. The arc starts on the ellipse from the angle **sa**, to the angle **ea**. The angles are given in 360 degree to **INT_MAX** ratio form. Uses the current color and mode in output surface file **f**.

void pa_fchord(FILE* f, int x1, int y1, int x2, int y2, int sa, int ea);

Draw a filled cord. Draws a filled cord on an ellipse, whose bounding box is **x1**,**y1** to **x2**,**y2**. The cord starts on the ellipse from the angle **sa**, to the angle **ea**. The angles are given in 360 degree to **INT_MAX** ratio form. Uses the current color and mode in output surface file **f**.

void pa_ftriangle(FILE* f, int x1, int y1, int x2, int y2, int x3, int y3);

Draw a filled triangle. Draws a filled triangle given the three points **x1**,**y1**,**x2**,**y2**, and **x3**,**y3**. Uses the current color and mode in output surface file **f**.

void pa_cursorg(FILE* f, int x, int y);

Position the cursor graphically in output surface file **f**. Moves the cursor to the pixel position **x** and **y**.

int pa_baseline(FILE* f);

Find baseline of current font. Finds the baseline, or line on which all characters of the current font sit upon, in terms of offset from the character bounding box origin in y in output surface file **f**.

void pa_setpixel(FILE* f, int x, int y);

Set single pixel. Sets a single pixel at the point **x**,**y**, using the current color and mode in output surface file **f**.

void pa_fover(FILE* f);

Set overwrite mode foreground. Sets all new drawing to overwrite old colors on the foreground in output surface file **f**.

void pa_bover(FILE* f);

Set overwrite mode background. Sets all new drawing to overwrite old colors on the background in output surface file **f**.

void pa_finvis(FILE* f);

Set invisible mode foreground. Sets all new drawing to discard colors on the foreground in output surface file **f**.

void pa_binvis(FILE* f);

Set invisible mode background. Sets all new drawing to discard colors on the background in output surface file **f**.

void pa_fxor(FILE* f);

Set xor mode foreground. Sets all new drawing to xor old colors with new colors on the foreground in output surface file **f**.

void pa_bxor(FILE* f);

Set xor mode background. Sets all new drawing to xor old colors with new colors on the background in output surface file **f**.

void pa_fand(FILE* f);

Set and mode foreground. Sets all new drawing to and old colors with new colors on the foreground in output surface file **f**.

void pa_band(FILE* f);

Set and mode background. Sets all new drawing to and old colors with new colors on the background in output surface file **f**.

void pa_for(FILE* f);

Set or mode foreground. Sets all new drawing to or old colors with new colors on the foreground in output surface file **f**.

void pa_bor(FILE* f);

Set or mode background. Sets all new drawing to or old colors with new colors on the background in output surface file **f**.

int pa_chrsizx(FILE* f);

Find character size in x in output surface file **f**. Returns the x size, in pixels, of the characters in the current font. If the font is proportional, its x size will vary per character. The size will then be the space character, which is guaranteed to be the widest character in the font.

int pa_chrsizy(FILE* f);

Find character size in y in output surface file **f**. Returns the y size, in pixels, of the characters in the current font.

int pa_fonts(FILE* f);

Find number of fonts in output surface file **f**. Returns the number of fonts installed on the system.

void pa_font(FILE* f, int fc);

Select logical font in output surface file **f**. Selects a font by logical number **fc**, where **fc** is 1..**pa_fonts()**.

void pa_fontnam(FILE* f, int fc, char* fns, int fnsl);

Find name of logical font in output surface file **f**. Returns the name of the logical font **fc** in the string **fns**. The length of the buffer is passed in **fnsl**. The buffer must be long enough to include the whole string and a zero termination.

void pa_fontsiz(FILE* f, int s);

Set font size in output surface file **f**. Sets the height of the current font, in pixels.

void pa_chrspcy(FILE* f, int s);

Set character y spacing in output surface file **f**. Sets the line to line extra spacing to **s**, in pixels.

void pa_chrspcx(FILE* f, int s);

Set character x spacing in output surface file **f**. Sets the character to character extra spacing **s**, in pixels.

int pa_dpmx(FILE* f);

Find dots per meter x. Finds the dots per meter in x of the current display device in output surface file **f**.

int pa_dpmy(FILE* f);

Find dots per meter y. Finds the dots per meter in y of the current display device in output surface file **f**.

int pa_strsiz(FILE* f, const char* s);

Find pixel size of string in output surface file **f**. Finds the total x size of the string **s**, in pixels. Accounts for all sizes and spacing.

int pa_chrpos(FILE* f, const char* s, int p);

Find pixel offset of character in output surface file **f**. Finds the pixel offset from the start of a string **s** in terms of x pixels, to the character by the index **p**.

void pa_writejust(FILE* f, const char* s, int n);

Write string justified in output surface file **f**. Writes the given string s into the number of x pixels n. If the space is more than is required, the extra space will be distributed between the characters. If the space given is insufficient for the characters to be drawn, it will be drawn in the minimum amount of space for the entire string.

int pa_justpos(FILE* f, const char* s, int p, int n);

Find justified pixel off set of character in output surface file **f**. Finds the pixel offset from the start of a string **s**, in terms of x pixels, to the character by the index **p**, for a string justified to fit into **n** pixels. The rules of justification are the same as for **pa_writejust()**.

void pa_condensed(FILE* f, int e);

Set condensed mode in output surface file **f**. Sets the current font to occupy a shorter baseline than normal. Note that this effect may not be implemented.

void pa_extended(FILE* f, int e);

Set extended mode in output surface file **f**. Sets the current font to occupy a longer baseline than normal. Note that this effect may not be implemented.

void pa_xlight(FILE* f, int e);

Set extra light mode in output surface file **f**. Sets the current font to extra light printing. Note that this effect may not be implemented.

void pa_light(FILE* f, int e);

Set light mode in output surface file **f**. Sets the current font to light printing. Note that this effect may not be implemented.

void pa_xbold(FILE* f, int e);

Set extra bold mode in output surface file **f**. Sets the current font to extra bold printing. Note that this effect may not be implemented.

void pa_hollow(FILE* f, int e);

Set hollow mode in output surface file **f**. Sets the current font to hollow, or sunken look, printing. Note that this effect may not be implemented.

void pa_raised(FILE* f, int e);

Set raised mode in output surface file **f**. Sets the current font to raised, or relief look, printing. Note that this effect may not be implemented.

void pa_settabg(FILE* f, int t);

Set graphical tab in output surface file **f**. Sets a tab to the pixel **t**.

void pa_restabg(FILE* f, int t);

Reset graphical tab in output surface file **f**. The tab at pixel **t** is removed. If no tab is set at **t**, no error will result.

void pa_fcolorg(FILE* f, int r, int g, int b);

Set foreground color graphical in output surface file **f**. The foreground color is set to the red **r**, green **g** and blue **b** color. The colors are in 0..**INT_MAX** ratios, with 0 = black, and **INT_MAX** = saturated. The nearest color to the one given is found and set active as the foreground color. The exact color that results will depend on the total number of colors implemented in the system, and could well be black and white in a system so equipped.

void pa_fcolorc(FILE* f, int r, int g, int b);

Set foreground color character in output surface file **f**. The foreground color is set to the red **r**, green **g** and blue **b** color. The colors are in 0..**INT_MAX** ratios, with 0 = black, and **INT_MAX** = saturated. The nearest color to the one given is found and set active as the foreground color. The exact color that results will depend on the total number of colors implemented in the system, and could well be black and white in a system so equipped.

This function gives you direct setting of character colors in RGB format if your system is so capable.

void pa_bcolorg(FILE* f, int r, int g, int b);

Set background color graphical in output surface file **f**. The background color is set to the red **r**, green **g** and blue **b** color. The colors are in 0..**INT_MAX** ratios, with 0 = black, and **INT_MAX** = saturated. The nearest color to the one given is found and set active as the foreground color. The exact color that results will depend on the total number of colors implemented in the system, and could well be black and white in a system so equipped.

void pa_bcolorc(FILE* f, int r, int g, int b);

Set background color character in output surface file **f**. The background color is set to the red **r**, green **g** and blue **b** color. The colors are in 0..**INT_MAX** ratios, with 0 = black, and **INT_MAX** = saturated. The nearest color to the one given is found and set active as the foreground color. The exact color that results will depend on the total number of colors implemented in the system, and could well be black and white in a system so equipped.

This function gives you direct setting of character colors in RGB format if your system is so capable.

void pa_loadpict(FILE* f, int p, char* fn);

Load picture into output surface file **f**. Loads the picture from the filename **fn** to logical picture number **p**, which is 1..n. The file must be in a format that the system understands, and is converted to an in memory format that is optimal for the system, such as a direct match for the graphics device in use.

int pa_pictsizx(FILE* f, int p);

Find picture size x in output surface file **f**. Returns the size, in pixels of x, of the logical picture **p**.

int pa_pictsizy(FILE* f, int p);

Find picture size y in output surface file **f**. Returns the size, in pixels of y, of the logical picture **p**.

void pa_picture(FILE* f, int p, int x1, int y1, int x2, int y2);

Draw picture into output surface file **f**. The logical picture **p** is drawn into the bounding box formed by **x1**,**y1** to **x2**,**y2**. The picture is stretched or compressed as required to fix into the destination bounding box. The method used to stretch or compress may depend on how much computing power is available on the system. There is no attention paid to aspect ratio. If the aspect ratio is to be preserved, it must be calculated. The current foreground mode is used.

void pa_delpict(FILE* f, int p);

Remove logical picture from use in output surface file **f**. The logical picture **p** is removed from the picture queue, and will no longer take up memory space.

void pa_scrollg(FILE* f, int x, int y);

Scroll in arbitrary directions in output surface file **f**. The screen is scrolled according to the differences in **x** and **y**, which are in pixels. Uncovered areas on the screen appear in the current background color.

void blockcopyg(FILE* f, int s, int d, int sx1, int sy1, int sx2, int sy2, int dx1, int dy1, int dx2, int dy2);

Copy a pixel block between buffers in output surface file **f**. Copies a block of pixels from the source buffer **s** with the bounding box **sx1**, **sy1** to **sx2**, **sy2** to the destination buffer **d** in bounding box **dx1**, **dy1** to **dx2**, **dy2**. The current foreground write mode is used. The picture is stretched or compressed as required to fit into the destination bounding box. The method used to stretch or compress may depend on how much computing power is available on the system. There is no attention paid to aspect ratio. If the aspect ratio is to be preserved, it must be calculated. The current foreground mode is used.

9.23 Events In graphics

See the description of the event record (9.21 “Declarations”) for the format of the entire record. **graphics** events carry over the set of events from **terminal**, and add only one new one.

Event: pa_etmoumovg

The mouse with handle **mmoung** has moved, to the graphical position indicated by **moupxg** and **moupyg**.

1.1

10 Windows Management Library

windows is completely upward compatible with **terminal** and **graphics**. Given a single fixed screen, **windows** subdivides the screen into virtual windows, which can be set to any size or position. A **terminal** compatible program sees its window as a "virtual screen", and does not know that some or all of the contents of that screen may be hidden. A **windows** aware program can participate in the benefits of a windowed environment.

10.1 Screen Appearance

The idea of windows management is to take a program that thinks it is talking to an ordinary terminal, and allow the presentation of multiple such windows within a single screen.

By default, **windows** emulates a **terminal** interface for each window that is identical to the behavior of a full screen window from the programs' point of view. A standard size screen is implemented for the program of 80 characters by 25 lines (even if the real display has a different size). **windows** accepts program writes, and places that information in a buffer that looks like an ideal screen. Then, the contents of that buffer are placed on the screen in various arrangements to complete the desktop for the user.

10.2 Window Modes

A window can be any actual size on the display. A window can also be off the display entirely, so that it accepts changes to its buffer, but does not display those changes. In addition, a window can be active, inactive, or overlapped, or even completely covered by other windows on the display.

10.3 Buffered Mode

A window comes up by default in the "buffered" mode. In buffered mode, the program has a "virtual display surface" that it writes to. **windows** maintains this surface in memory, and maintains the actual onscreen window as a scrolled window on that. The program is unconcerned with what part of its screen is being displayed, or even if it is displayed at all. The user operates the window using the frame controls.

Buffered mode is designed to allow the program to be unconcerned with the management of the display. However, the program can set the size of the virtual display using **pa_sizbuf[g](f, x, y)**. When a buffer is resized, its contents is cleared to the current background color.

The buffer is a display surface that the buffer handler keeps for the program. The buffered mode allows a program to "think" it has a window of size N, but the appearance of the window on the actual display is a window on that virtual display that is maintained by the buffer handler.

Because from the programs point of view the window size does not change, and is always a complete window, the resize events are performed transparently to the program. Similarly, the minimize and maximize events are handled for the program (see 10.4 “Unbuffered Mode”). Discovery is when a window or part of a window is uncovered on the display.

When in buffered mode, the window manager is may use scroll bars to display within a screen view that is smaller than the buffer.

Although buffered mode automatically handles window status events (covered in 10.4 “Unbuffered Mode”), these events are still sent through. For maximum compatibility, the using program should ignore any events not relevant to the mode it is in.

10.4 Unbuffered Mode

The program can leave buffered mode by using **pa_buffer(f, b)**. Unbuffered mode has no display buffer, and no default actions for events. Instead, the onscreen appearance of the window is set dynamically by the program.

When unbuffered mode is entered, the buffer for the window is discarded, and it becomes entirely up to the program to manage its own window by watching events. The size of the window no longer reflects the buffer, but instead is the size of the onscreen window. In contrast to buffered mode, this can change many times as the window is resized under user control.

Buffered mode can be reentered at any time. The contents of the buffer are cleared to spaces, or the background color.

When running unbuffered, it is assumed that the program will manage the update of the on-screen display surface itself. When running unbuffered, the program handles events that are normally handled automatically.

The **pa_etredraw** event contains the limits of an update rectangle that shows what part of the client area needs to be redrawn on the screen. The program can ignore the update rectangle, and simply redraw the entire screen, or it may only redraw the figures that overlap the update rectangle. The latter is usually more time efficient. **pa_etredraw** gives its update rectangle in character coordinates. There is also a pixel version of that, **pa_etredrawg**, which uses pixel coordinates. On a graphical system, both messages will be sent, and they duplicate each other. The character coordinate **pa_etredraw** will contain a rectangle of characters that at least covers the pixels that need to be redrawn in a graphical system. Since **pa_etredraw** and **pa_etredrawg** duplicate each other, only one of them should be obeyed, and the other ignored.

The **pa_etresize** event indicates that the onscreen window has changed its size. Its purpose is to let the program update any calculations based on the size of the window. This event can be ignored, used to update stored **pa_maxx[g]** and **pa_maxy[g]** values, or it could even cause the entire arrangement of the screen to be reformed.

pa_etresize is not normally used to redraw areas of the screen. If a resize event causes new screen area to be uncovered, then one or more **pa_etredraw** events will also be sent. There is no way to determine exactly which redraw operations correspond to the resize event, so some redundancy is possible with applications that repack their window's contents on a resize.

The **pa_etmin** event indicates that the window has been minimized. When a window is minimized, it will not be displayed, and won't receive **pa_etredraw** events. On the desktop, the window is represented by a small icon. An alternative form of minimization is for the window to receive an **pa_etmin** message, followed by a **pa_etresize** message, and continued **pa_etredraw** messages. This a hint for the window to display vital information in a much smaller format that fits into the minimized icon.

The **pa_etmax** event indicates that the window has been maximized, or covers the entire desktop. If the window needs to be updated, this event may be accompanied by **pa_etresize** and **pa_etredraw** events.

The **etnorm** event indicates a normal window mode has been resumed from **pa_etmin** or **pa_etmax**.

1.1

In many modern computers, the screen buffers are in fast video memory and managed by graphics hardware. Thus it can be more efficient for the underlying system to process all writes to an offscreen buffer and then copy back to the user screen. Thus the unbuffered mode may in fact have no real effect. The main difference for the client program is that **pa_etredraw** events are rarely, if ever sent, since the system has a copy of all onscreen graphics.

10.5 Buffer Follow Mode

window has an intermediate mode between buffered and unbuffered mode. With faster computers and more memory, programs often retain the screen buffers and simply resize them as the window is resized. To allow for this mode, the **pa_etresize** message also contains **rszx**, **rszy**, **rszgx** and **rszgy** parameters that have the new size of the window. These parameters are normally ignored. For buffer follow mode, the **rszx**, **rszy**, **rszgx** and **rszgy** parameters are used to resize the buffer using **pa_sizbuf[g](f,x,y)**. The new buffer sizes are reflected in **pa_maxx[g](f)** and **pa_maxy[g](f)** and the program can continue to use the screen buffers as before. Note that the **pa_sizbuf[f](f,x,y)** call resizes all of the screen buffers and clears them to the background color.

Buffer mode gives client programs the ability to treat the screen as a virtual screen whose geometry does not change. The **pa_etresize** message, which is normally ignored in buffered mode, gives the status of what the actual display window is doing, and the client program has the option of reconfiguring the buffer to “follow” what the display window is doing.

10.6 Defacto transparency

Using unbuffered mode, it is possible to achieve window transparency. Transparency means that some parts of the window will show through to the display surface underneath the window. This can be used to implement advanced effects like non-rectangular windows, and it is the basis for widgets (which appear in 11 “Widget Library”). For defacto transparency to work, the drawing order of windows must be tightly controlled.

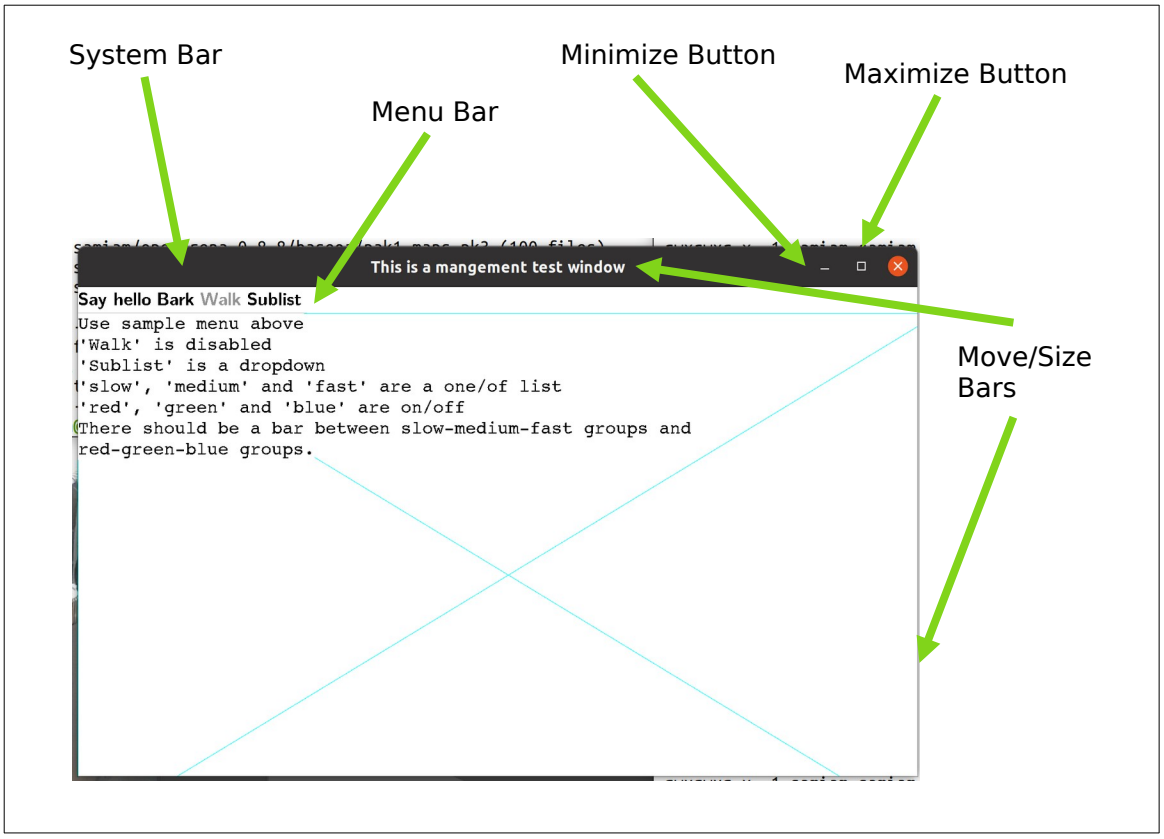
10.7 Delayed Window Display

When a window is created, it is not displayed until an actual write operation is made to it. This allows the window to be changed in configuration by the program before it is displayed, and prevents the window being rapidly changed on screen as that happens. For example, the program might want to take the window out of buffered mode, change its size, remove the frame, or add menus.

10.8 Window Frames

Under most window systems, each window has a "window frame", which has various window management functions. These include:

- Move bars to resize and move the window.
- A title bar that indicates what program is running in the window.
- A "minimize" button.
- A "maximize" button.
- A "menu" bar.



A window has several system components, not all of which are visible. Here the move bars are invisible unless you move the mouse cursor across them.

When a window is in buffered mode, none of the frame features are under the control of the program, but instead are managed by the buffering software.

The appearance, or even the existence, of any of these items can be customized by the program. The frame can be enabled or disabled by **pa_frame(f, e),** where **f** is the window file, and **e** is true to enable the frame. The title can be set by **pa_title(f, e)**. The title, minimize and maximize buttons are considered part of the "system bar", which can be enabled or disabled by **pa_sysbar(f, e)**. The resize bars are enabled or disabled by **pa_sizable(f, e)**.

The frame control functions **pa_frame**, **pa_title**, **pa_sysbar** and **pa_sizable** are optional within an implementation. The system may not allow an application to, for example, disable its sizing bars, or it may not have a set of system controls on the window. The program must be ready for these commands not to have an effect. For example, after the **pa_sizable** function is set false, it should still be ready to receive a new size. If, for example, the size of the graphic to be presented is a fixed size, it would be filled out or clipped to fit the resulting window.

10.9 Multiple Windows

The **stdout** file is used as the output and the **stdin** file as input to the main window by default. A window can also be explicitly opened by **pa_openwin(inw,outw,par,id)**. When a new window is

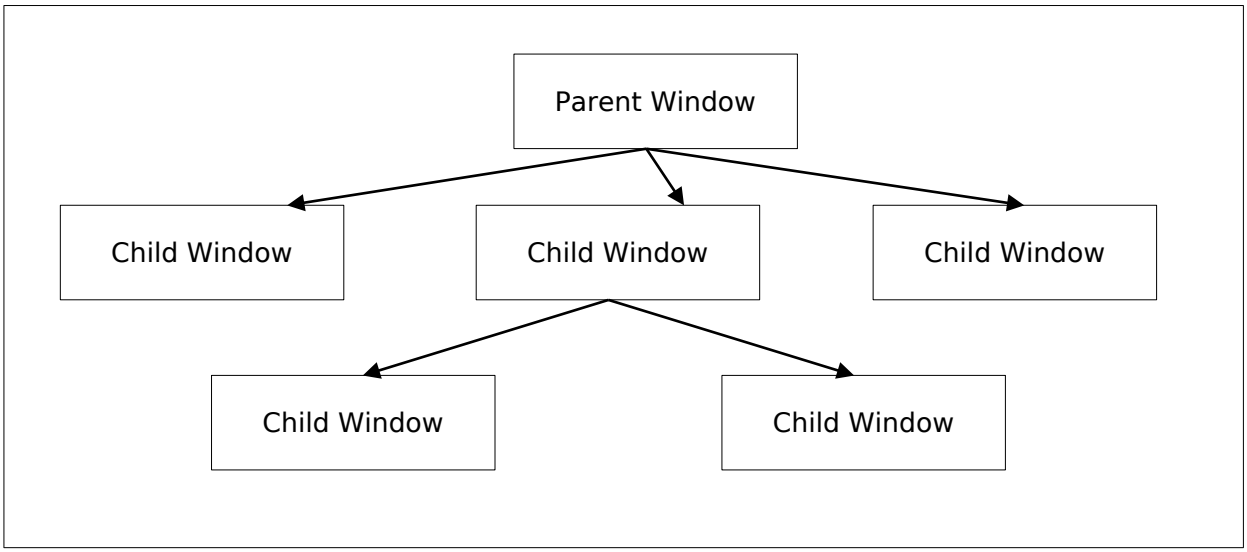
opened, it is placed on the desktop with the other windows. The files **inw** and **outw** specify the input and output files attached to the window. The input file receives all user input and events from the window, and all of the write and other drawing operations are sent to the output file. The **id** is a number, from 1 to N, that specifies the logical window. The logical window number 1 is reserved for the **stdin** and **stdout** pair. The **par** parameter is NULL for a parentless window.

The output side of a window must always be unique, but the input side can be shared. Multiple windows can be attached to a single input file, and that input file will receive all of the input and events from all attached windows. The logical window number is returned with each event, so that the source of the input or event can be determined. This forms the basis of “class window handling” (10.13 “Class Window Handling”), or using one input handler to handle multiple windows.

Windows can be closed using the standard ANSI C **fclose()** call. Only the output side of a window is used to close that window. The input side is automatically closed when there is no longer a window that references it.

10.10 Parent/Child Windows

Windows systems are tree structured, and each window is a child of a parent window, with the exception of a "root" or master window, which is usually the window that contains the entire desktop. The methods introduced create windows that live side by side on the desktop, and have the desktop as their parents. However, it is possible to nest windows within each other.



A window is opened as a child by **pa_openwin(inw, outw, par, id)**, which is the same **pa_openwin()** call with a parent specified. The parent file **par** is the text file that is attached to the output window of the parent. To be a child window means to move as one within it. It maintains its position within the parent, even as the parent itself moves. It always is in front of the Z ordering for the parent. It is minimized, maximized and closed with the parent.

All windows have their position given in relative terms to the parent's client area origin. Any position operation is also relative to the parent.

The common use of the parent is to create "sibling" windows, or windows equivalent to the default program window in status, that are positioned independently within the parent. A program starts as the child of an external parent window. This could be an actual program, such as a program acting as a manager, or it could be the desktop root. There is no difference between these for the program.

Programs do not have direct access to the parent window in which the program was created. That parent window is usually the desktop.

10.11 Moving and Sizing Windows

Moving a window is done with the **pa_setpos[g](f, x, y)** procedure. The position is given in parent coordinates. When a window is on the desktop, it is necessary to find the size of the desktop, which is found with **pa_scnsiz[g](f, x, y)**. When the desktop size is returned for character sizes, this is calculated using the current character sizes for the window. This is for comparative purposes only. The desktop may use a completely different set of characters and sizes. The purpose of giving the parent sizes is only to allow the program to determine the relative size and placement of the child window in the parent.

Sizing a window is done with **pa_setsiz[g](f, x, y)** procedure. This sets the size of the entire window, and so does not indicate the resulting size of its client area. To find out what window size is needed for a given client area, before actually sizing the window, the function **pa_winclient[g](f, cx, cy, wx, wy, ms)** is used. It returns the window size needed to contain that client. **pa_winclient[g]** takes a set of the modes of the window to enable it to make the size determination.

The mode set declaration is:

```
/* windows mode sets */
typedef enum {

    pa_wmframe, /* frame on/off */
    pa_wmsize,  /* size bars on/off */
    pa_wmsysbar /* system bar on/off */

} pa_winmod;
typedef int pa_winmodset;
```

To find the current size of a window, in parent terms, the procedure **pa_getsiz[g](f, x, y)** is used. This procedure is useful when you need to find the size of a window on the desktop.

10.12 Z Ordering

The Z ordering, or back to front ordering of windows, is determined by default to be newest created windows to the front, oldest to the back. This order can be modified by the users when they select windows towards the back to come to the front. The program can also reorder windows at will by sending them to the front or back. Note that the Z ordering is always relative within a parent.

To send a window to the front, the call **pa_front(f)** is used. To send a window to the back, the call **pa_back(f)** is used. To achieve a specific order within a set of windows, they should be send to the front in the opposite order from which they are to appear, IE., the backmost first, and the frontmost last. Alternately, the windows can be sent to the back in the order they appear.

1.1

There are other reasons besides reordering windows that these functions might be useful. When an error in encountered, it is common to send a window containing the error message to the front of the parent Z order, and to send the entire window to the front. Placing an active display or a background pattern in a frameless maximized window can create wallpaper. Placing the same window in the back can be a screen saver.

10.13 Class Window Handling

windows handles windows as a set of two files, the input and output. However, these don't need to always be a set. It is possible to reuse an already open input file as the input side of any new window. This is possible because the window id supplied when the window is opened is passed with all messages to the event procedure.

Allowing a single program thread to handle multiple open windows allows the creation of multiple window programs without the need to create a multitask program. For each message, the id is examined, and the action performed on the appropriate output file window.

Because the output file is unique, it is always the handle used to refer to the window in management calls.

10.14 Parallel Windows

Parallel tasking is a natural match to a multiwindowing system. With a task created to operate each window, the program is simplified, and user response is generally better.

To increase the responsiveness of user interfaces, the recommended program design for windowing programs is to create two tasks for each window, the "foreground" and the "background" tasks. The foreground task performs the event loop and the redraw, resize, move and other user interface tasks. The background task would handle computation tasks and other tasks related to performing actions or changing the drawing surface. The foreground task determines if an action that requires the background task is required, then sends a command to the background task via a queue or other IPC (InterProcess Communication).

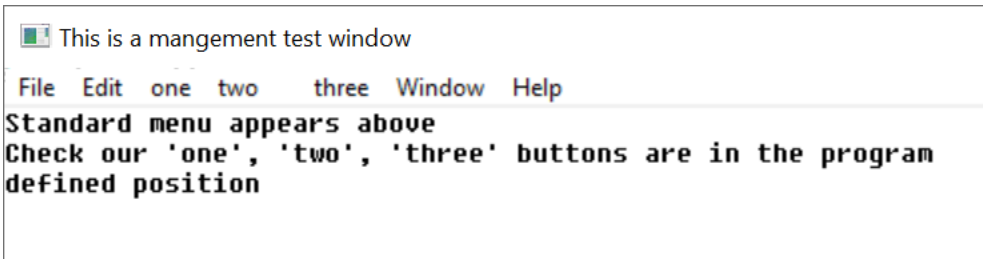
The reason for this construction is that the things the user sees, such as keeping the client area redrawn, or obeying a move, resize, minimize or similar command always have a task waiting to perform them. This means that elementary user desktop management appears responsive to the user, while data manipulation and computation tasks simply delay client area updates. Users will be much more willing to tolerate client area slowdowns than having a window apparently lock stubbornly in position. Ideally, the client area should also have an indication that computation is taking place. The most modern method for this is a progress indicator that gives estimates as to when a task will complete, if the task will take longer than about 1 second.

If the work performed in a client area is trivial, only a foreground task need be provided. But be aware that it is very easy to slip into a mode where essential updates are held off. Calling a file procedure, waiting on a timer, or other simple task compromises the response time for window management. Better to leave the window in buffered mode, or structure with foreground/background from program creation than to have to add it later.

10.15 Menus

The menu is a series of buttons labeled with text for user action. The buttons can either have a direct effect on the program, or they can activate a series of submenus in a tree structure.

The exact location of a menu is system dependent. It may or may not affect the client area.



A menu on a window.

A menu is described to **windows** by constructing a data structure:

```
/* menu */
typedef struct pa_menurec* pa_menuptr;
typedef struct pa_menurec {

        pa_menuptr next;   /* next menu item in list */
        pa_menuptr branch; /* menu branch */
        int        onoff;  /* on/off highlight */
        int        oneof;  /* "one of" highlight */
        int        bar;    /* place bar under */
        int        id;     /* id of menu item */
        char*      face;   /* text to place on button */

} pa_menurec;
```

The menu is activated by **pa_menu(f, m)** where **m** is the data structure for the menu.

Menu buttons are highlighted when pointed to. Additionally, a button can have "on/off" highlighting. In this mode, a state is kept for the button that is flipped on or off as the button is pressed. The highlighting for on and off states is different. The data structure that defines a menu is not used or updated after it is used to create a menu.

The **onoff** field true selects on/off highlighting. After the menu is activated, the state of the button can be changed with **pa_menusel(f, id, e)**. The button state is not automatically changed by a user menu select. The program must specifically change the state of the button.

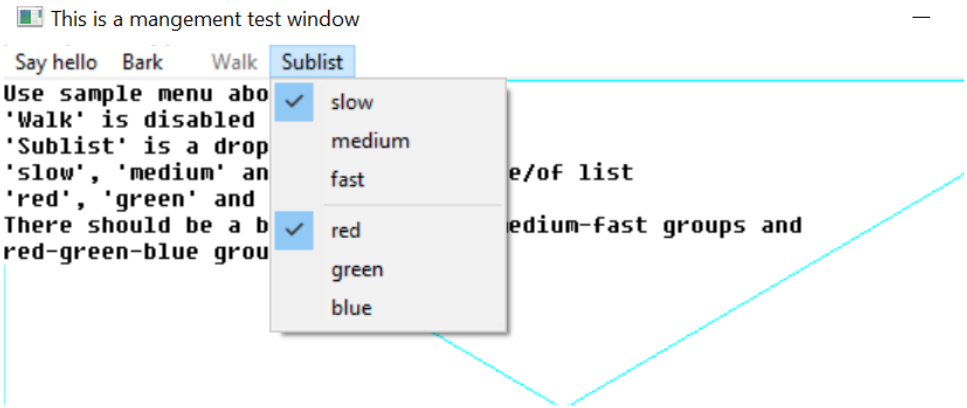
Another highlight mode is "one of" or "checklist" highlighting. In this case, a group of related buttons are joined by setting the **oneof** flag on each item in the list but the last. The highlighting for a button that is selected is different from one that is not, and only one item will be active in the list.

Like on/off buttons, user selection does not automatically change the state of the buttons in the list. It must be specifically changed by a **pa_menusel()** call.

Typically, neither on/off nor one/of switching is available for top level (horizontal) menus, and the program should not count on them.

1.1

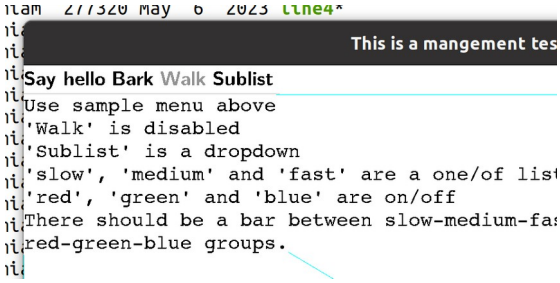
Menu items can be grouped in a vertical list by setting the **bar** field active. This causes a horizontal bar to be drawn under the menu item. Vertical lists are described next in "menu sublisting".



Selected menu items “slow” and “red”. Note bar between top and bottom groups.

A button can be enabled or disabled. A disabled button is highlighted specially, typically as greyed out. It indicates a button that is entirely inactive, because that function is not available. This is done to allow the same menu to be used regardless of the availability of the functions on the menu. Also, some functions come and go. For example, a "save" button (for "save file") might only be active if the file has changed, and needs to be saved.

The enable status can be changed after the menu is activated by **pa_menuena()**.



The “walk” menu item is disabled.

Each button in a menu has a numeric **id**. This is used by several functions to change the state of buttons. It is an error to have two buttons with the same id.

The **face** text string indicates what text will appear on the face of the button. The system will automatically size the buttons to fit the largest text of a button, so care should be taken not to have a single button's text so long as to cause a lot of white space in the menu.

10.16 Setting Menu Active

A menu is constructed by the program as a list using the menu data structure, then set active by calling **pa_menu()**, which can also turn the menu back off.

When a menu data structure is activated, all of the fields are copied and placed into an internal form. After the activation, the menu data has no function, and changing its fields will have no effect. The menu data structure can be recycled immediately after the **pa_menu()** call.

10.17 Setting Menu States

For an on/off button, the procedure **pa_menuena(f, id, e)** is used to enable or disable the button after it has been placed in a menu. For **oneof** highlighting, the function **pa_menusel()** is used. This removes any other select active, and either sets the given button active, or no button active.

10.18 Standard Menus

Besides presenting a menu in the method standard for the implementation, it is also likely that there exists a standard arrangement of commonly used buttons in the menu. **windows** defines a series of such standard buttons.

1.1

```
/* standardized menu entries */
#define PA_SMNEW         1 /* new file */
#define PA_SMOPEN        2 /* open file */
#define PA_SMCLOSE       3 /* close file */
#define PA_SMSAVE        4 /* save file */
#define PA_SMSAVEAS      5 /* save file as name */
#define PA_SMPAGESET     6 /* page setup */
#define PA_SMPRINT       7 /* print */
#define PA_SMEXIT        8 /* exit program */
#define PA_SMUNDO        9 /* undo edit */
#define PA_SMCUT        10 /* cut selection */
#define PA_SMPASTE      11 /* paste selection */
#define PA_SMDELETE     12 /* delete selection */
#define PA_SMFIND       13 /* find text */
#define PA_SMFINDNEXT   14 /* find next */
#define PA_SMREPLACE    15 /* replace text */
#define PA_SMGOTO       16 /* goto line */
#define PA_SMSELECTALL  17 /* select all text */
#define PA_SMNEWWINDOW  18 /* new window */
#define PA_SMTILEHORIZ  19 /* tile child windows horizontally */
#define PA_SMTILEVERT   20 /* tile child windows vertically */
#define PA_SMCASCADE    21 /* cascade windows */
#define PA_SMCLOSEALL   22 /* close all windows */
#define PA_SMHELPTOPIC  23 /* help topics */
#define PA_SMABOUT      24 /* about this program */
#define PA_SMMAX        24 /* maximum defined standard menu entries */

/* standard menu selector */
typedef int pa_stdmenusel;
```

Standard menu button definitions should be used whenever possible. To create a menu using standard menu buttons, the procedure **pa_stdmenu()** is used. The arrangement of the menu is chosen to match the implementation, and the non-standard buttons provided are integrated into the menu in the normal place for the implementation. When standard buttons are used, the button ids are set to the values shown above, so these values should not be used by the program.

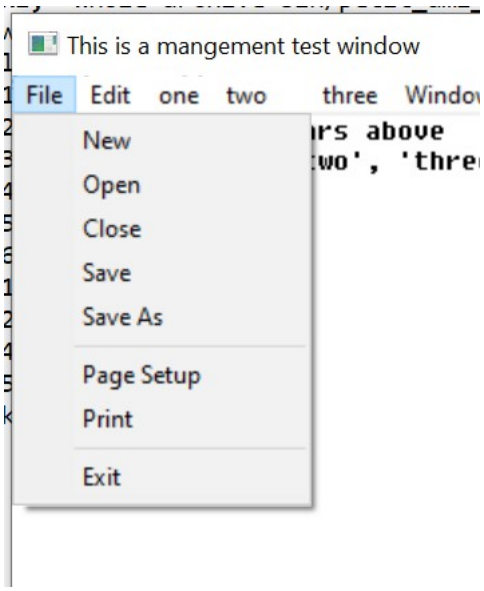
Note that **pa_stdmenu()** simply constructs a menu, it does not set it active.

10.19 Menu Sublisting

When there is not enough room to represent the entire menu on a window, or anytime the size of a menu must be compressed, the tree structure of menus can be used with "pulldown" menus. A pulldown menu is a vertical menu list that appears under the selected button. This is done by placing a submenu in the **branch** pointer of the menu structure. The branch defines the pulldown menu items. The branch menu item is specially indicated to show that it is a branch, and not an immediate button, usually with an arrow pointing towards where the branch pulldown will appear. Any number of levels of pulldown menus can be created. The levels under the topmost branch are generally placed to the right of the branch button.

If a branch pulldown cannot be placed where it should, due to the proximity of the button to the edge of the display, it instead goes back in the other direction, up or down, or both, until space is found for it. If the entire pulldown cannot be displayed, it is truncated. This would only happen if the pulldown was too large for the display.

When a branch button is pressed, no event is generated for it. The button is strictly used to activate the pulldown.



A pulldown menu.

10.20 Advanced Windowing

The features of a window besides the client area are designed to be the minimum features implemented across platforms. To implement more advanced windows appearances with custom menus, controls and status lines, the standard method is to turn off frames and menus for child windows, and use them as building blocks for more advanced client areas with multiple subwindows and features.

It is recommended that at least one menu appear for a program, even if the functions duplicate some of the advanced controls provided. The reason for this is that the menu has different presentation methods in different systems, so providing the standard "top" menu will help programs' portability.

10.21 Events

New event types are added for the management mode. As usual, events beyond the definition here should be ignored.

```
/* events */
typedef enum {
    pa_etchar,      /* ANSI character returned */
    pa_etup,        /* cursor up one line */
    pa_etdown,      /* down one line */
    pa_etleft,      /* left one character */
    pa_etright,     /* right one character */
```

```
    pa_etleftw,    /* left one word */
    pa_etrightw,   /* right one word */
    pa_ethome,     /* home of document */
    pa_ethomes,    /* home of screen */
    pa_ethomel,    /* home of line */
    pa_etend,      /* end of document */
    pa_etends,     /* end of screen */
    pa_etendl,     /* end of line */
    pa_etscrl,     /* scroll left one character */
    pa_etscrr,     /* scroll right one character */
    pa_etscru,     /* scroll up one line */
    pa_etscrd,     /* scroll down one line */
    pa_etpagd,     /* page down */
    pa_etpagu,     /* page up */
    pa_ettab,      /* tab */
    pa_etenter,    /* enter line */
    pa_etinsert,   /* insert block */
    pa_etinsertl,  /* insert line */
    pa_etinsertt,  /* insert toggle */
    pa_etdel,      /* delete block */
    pa_etdell,     /* delete line */
    pa_etdelcf,    /* delete character forward */
    pa_etdelcb,    /* delete character backward */
    pa_etcopy,     /* copy block */
    pa_etcopyl,    /* copy line */
    pa_etcan,      /* cancel current operation */
    pa_etstop,     /* stop current operation */
    pa_etcont,     /* continue current operation */
    pa_etprint,    /* print document */
    pa_etprintb,   /* print block */
    pa_etprints,   /* print screen */
    pa_etfun,      /* function key */
    pa_etmenu,     /* display menu */
    pa_etmouba,    /* mouse button assertion */
    pa_etmoubd,    /* mouse button deassertion */
    pa_etmoumov,   /* mouse move */
    pa_ettim,      /* timer matures */
    pa_etjoyba,    /* joystick button assertion */
    pa_etjoybd,    /* joystick button deassertion */
    pa_etjoymov,   /* joystick move */
    pa_etresize,   /* window was resized */
    pa_etterm,     /* terminate program */
    pa_etframe,    /* frame sync */
    pa_etmoumovg,  /* mouse move graphical */
    pa_etredraw,   /* window redraw character */
    pa_etredrawg,  /* window redraw graphical */
    pa_etmin,      /* window minimized */
    pa_etmax,      /* window maximized */
    pa_etnorm,     /* window normalized */
```

```
    pa_etmenus,    /* menu item selected */
    pa_etbutton,   /* button assert */
    pa_etchkbox,   /* checkbox click */
    pa_etradbut,   /* radio button click */
    pa_etsclull,   /* scroll up/left line */
    pa_etscldrl,   /* scroll down/right line */
    pa_etsclulp,   /* scroll up/left page */
    pa_etscldrp,   /* scroll down/right page */
    pa_etsclpos,   /* scroll bar position */
    pa_etedtbox,   /* edit box signals done */
    pa_etnumbox,   /* number select box signals done */
    pa_etlstbox,   /* list box selection */
    pa_etdrpbox,   /* drop box selection */
    pa_etdrebox,   /* drop edit box signals done */
    pa_etsldpos,   /* slider position */
    pa_ettabbar    /* tab bar select */
} pa_evtcod;
/* event record */
typedef struct {

    /* identifier of window for event */ int winid;
    /* event type */                      pa_evtcod etype;
    /* event was handled */               int handled;
    union {

        /* these events require parameter data */

        /** etchar: ANSI character returned */  char echar;
        /** ettim: timer handle that matured */ int timnum;
        /** etmoumov: */
        struct {

            /** mouse number */   int mmoun;
            /** mouse movement */ int moupx, moupy;

        };
        /* etmouba */
        struct {

            /** mouse handle */  int amoun;
            /** button number */ int amoubn;

        };
        /* etmoubd */
        struct {

            /** mouse handle */  int dmoun;
            /** button number */ int dmoubn;
```

```
};
/* pa_etjoyba */
struct {

    /** joystick number */ int ajoyn;
    /** button number */   int ajoybn;

};
/* pa_etjoybd */
struct {

    /** joystick number */ int djoyn;
    /** button number */   int djoybn;

};
/* pa_etjoymov */
struct {

    /** joystick number */      int mjoyn;
    /** joystick coordinates */ int joypx, joypy, joypz;

};
/** pa_etresize */
Struct {

    Int rszx, rszy, rszxg, rszyg;

}
/* pa_etfun */
/** function key */ int fkey;
/** etmoumovg: */
struct {

    /** mouse number */   int mmoung;
    /** mouse movement */ int moupxg, moupyg;

};
/** etredraw, etredrawg */
struct {

    /** bounding rectangle */
    int rsx, rsy, rex, rey;

};
/* pa_etmenus */
int menuid; /* menu item selected */
/* pa_etbutton */
int butid; /* button id */
/* pa_etchkbox */
```

```
int ckbxid; /* checkbox */
/* pa_etradbut */
int radbid; /* radio button */
/* pa_etsclull */
int sclulid; /* scroll up/left line */
/* pa_etscldrl */
int scldlid; /* scroll down/right line */
/* pa_etsclulp */
int sclupid; /* scroll up/left page */
/* pa_etscldrp */
int scldpid; /* scroll down/right page */
/* pa_etsclpos */
struct {

    int sclpid; /* scroll bar */
    int sclpos; /* scroll bar position */

};
/* pa_etedtbox */
int edtbid; /* edit box complete */
/* pa_etnumbox,   /* number select box signals done */
struct {

    int numbid; /* num sel box select */
    int numbsl; /* num select value */

};
/* pa_etlstbox */
struct {

    int lstbid; /* list box select */
    int lstbsl; /* list box select number */

};
/* pa_etdrpbox */
struct {

    int drpbid; /* drop box select */
    int drpbsl; /* drop box select */

};
/* pa_etdrebox */
int drebid; /* drop edit box select */
/* pa_etsldpos */
struct {

    int sldpid; /* slider position */
    int sldpos; /* slider position */
```

1.1

```
        };
        /* pa_ettabbar */
        struct {

            int tabid;  /* tab bar */
            int tabsel; /* tab select */

        };

     };

} pa_evtrec, *pa_evtptr;
```

10.22 Procedures and Functions in windows

void pa_openwin(FILE** infile, FILE** outfile, FILE* parent, int wid);

Opens a new window. The input file **infile** will get messages pertaining to the window, and the output file **outfile** will be used to draw to it. The input file may already be open elsewhere, in which case it will get all messages for all open windows opened with it. If a previous **infile** is not used, it must be NULL. The window will be placed at 1,1 within the parent and held out of display. The output file is always used as the window handle, since it is unique to the window.

The window identifier **id** is a number from 1 to n that is returned in messages concerning the window.

Only the output side of a window pair is used in procedures or functions that operate on the window once opened with **pa_openwin()**.

A window is closed with the standard ANSI C close procedure, used on the output file for the window. The input side of the window is automatically closed when there are no longer any windows that reference it.

If the optional parent window is specified, the new window is created as a child of the given parent, otherwise it must be NULL. This means that it will always be displayed within the parent, and be clipped to it.

void pa_buffer(FILE* f, int e);

Engages or removes window **f** from buffered mode, according to the boolean **e**. In buffered mode, all of the drawing for a window is performed on a memory buffer, then copied to the screen. The screen view of the buffer can be all or part of the buffer, and multiple buffers can be managed.

If the buffer mode is enabled, the size of the buffer is set to the previous size.

void pa_sizbuf[g](FILE* f, int x, int y);

Sets the size of the buffer used to draw into. **x** and **y** indicate the width and height, respectively, of the buffer surface in window **f**. It is an error if buffering is not enabled. **pa_sizbuf()** sets the buffer size in characters. **pa_sizbufg()** sets the buffer size in pixels.

void pa_title(FILE* f, char* ts);

Sets the title of the window **f** to the string **ts**. If the title is too long for the current window size, an implementation defined method will be used to make it fit, for example, it is clipped.

void pa_frame(FILE* f, int e);

Enables or disables the appearance of the frame in window **f**, according to Boolean **e**, which includes the minimize, maximize, title, size, move and close controls. If the frame is removed, the user will be unable to operate the frame controls.

void pa_sysbar(FILE* f, int e);

Enables or disables the system control bar for a window **f**. If **e** is true, the system bar is enabled, otherwise disabled. The system bar normally includes the title, minimum, maximum, and other control buttons. If the frame is not enabled, then the system bar will not appear regardless of the **pa_sysbar()** status, but the size of it will be recorded.

void pa_sizable(FILE* f, int e);

Enables or disables the sizing bars for a window **f**. If **e** is true, the sizing bar is enabled, otherwise disabled. If the frame is not enabled, then the size bars will not appear regardless of the sizable status, but it will be recorded. If the size bars are removed, the user will be unable to resize the window.

void pa_setpos[g](FILE* f, int x, int y);

Sets the position, within the parent, of a child window **f**, using position **x** and **y**. If the window is on the desktop, then the window position is relative to the desktop. The **pa_setpos()** procedure sets the position in terms of characters, and the **pa_setposg()** procedure set the position in terms of pixels.

The position is set in terms of the parent's coordinates. The mode of the parent's coordinates may not be known. For example, the desktop may not even have a character mode, so the idea of a character size may be arbitrary. What matters in this case is the relative position and size in the desktop as determined by **pa_scnsiz()**.

void pa_scnsiz[g](FILE* f, int* x, int* y);

Finds the size of the user screen or desktop for window **f** to the size **x** and **y**. **pa_scnsiz()** returns the size in character terms, and **pa_scnsizg()** returns the size in pixel terms. If the desktop does not have a character mode, an arbitrary scale is created. What is important is the relative location within the desktop.

void pa_winclient[g](FILE* f, int cx, int cy, int* wx, int* wy, pa_winmodset ms);

Determines the window size needed for a given client size within window **f**. Given a desired client size of **cx** and **cy**, in width and height, the necessary window size to achieve that will be returned in **wx** and **xy**. **pa_winclient()** determines these measurements in character dimensions, and **pa_winclientg()** determines them in pixel terms.

The set of modes **ms** is used to determine the needed window size.

If the parent of the window has no character mode, then one is created that will be acceptable to **pa_setpos()**.

void pa_getsiz[g](FILE* f, int* x, int* y);

Finds the size of a window in parent coordinate terms for window **f**, to size **x** and **y**. **pa_getsiz()** returns the character size, and **pa_getsizg()** returns the pixel size.

If the parent has no character mode, then one is created that is compatible with other **windows** functions and procedures.

void pa_setsiz[g](FILE* f, int x, int y);

Sets the size of window **f** in parent coordinate terms, to size **x** and **y**. **pa_setsiz()** sets the character size, and **pa_setsizg()** sets the pixel size.

If the parent has no character mode, then one is created that is compatible with other **windows** functions and procedures.

void pa_back(FILE* f);

Sends the window **f** to the back of the parent Z order.

void pa_front(FILE* f);

Sends the window **f** to the front of the parent Z order.

void pa_menu(FILE* f, pa_menuptr m);

Sets up the menu bar for window **f** from the given list **m**, which contains a menu bar definition structure.

If the menu pointer is NULL, then the menu is removed.

The menu data structure is copied during the call, so the menu structure can be reused or freed.

void pa_stdmenu(pa_stdmenusel sms, pa_menuptr* sm, pa_menuptr pm);

Constructs a standard menu and returns that in **sm**. **sms** contains the set of desired standard buttons. **pm** contains a menu containing non-standard buttons to be added to the menu. The menu is constructed using the desired standard buttons, and the non-standard buttons placed into the menu at a standard location.

If **pm** contains no menu items, it should be NULL.

void pa_menuena(FILE* f, int id, int onoff);

Enables or disables an on/off menu button for window **f**. **id** refers to a button **id** that was specified in the menu data structure. If **e** is true, the button is enabled, otherwise disabled. The highlighting of the button will change to match.

void pa_menusel(FILE* f, int id, int select);

Selects a button from a **oneof** list to be active in window **f**. **id** refers to a button id that was specified in the menu data structure. If **e** is true, the button is selected, otherwise deselected. All other buttons in the **oneof** list are deactivated. The highlighting of the buttons will change to match.

1.1

10.23 Events In windows

See the description of the event record (10.21”Events”) for the format of the event record.

Event: pa_etresize

The window was resized. The new size in parent terms can be found with **pa_getsiz(f, x, y)**.

Event: pa_etredraw

Signifies that the rectangle represented by the starting point in the upper left hand corner (**rsx**, **rsy**) and the ending point in the lower right hand corner (**rex**, **rey**) should be redrawn. This redraw can be satisfied by either redrawing just the rectangle, or by redrawing the entire window client area.

It is possible that a complex area needing to be redrawn could be sent as a series of redraw commands.

Event: pa_etmin

Signifies that the window was minimized. This may not need any action, but is simply for information.

Event: pa_etmax

Signifies that the window was maximized. This may not need any action, but is simply for information. If the window needs to be redrawn as a result of the change, a separate redraw event will be sent

Event: pa_etnorm

The window was normalized back to its original size. This may not need any action, but is simply for information. If the window needs to be redrawn as a result of the change, a separate redraw event will be sent.

Event: pa_etmenus

A menu item was selected. The **id** parameter gives which menu item was selected, which is a logical identifier number selected when the menu structure was constructed.

11 Widgets Library

terminal and **graphics** define terminal and graphical operations on a fixed screen. **windows** defines its division into "virtual windows". **widgets** provides elements placed within those windows to allow user control. These include buttons, sliders, scroll bars, checkboxes, and similar user interface elements.

These are sometimes referred to as controls or widgets. Dialogs are predefined windows with widgets in them. What these user elements have in common is they all use **windows** elements to define the window they appear in, and **graphics** or **terminal** routines to draw their appearance.

widgets can be performed entirely in terms of **windows** with **graphics** or **terminal** calls. However, it is still dependent on a particular operating system because of its appearance. **widgets** maintains the "look and feel" of a particular operating system.

11.1 Tiles, Layers and Looks

Each widget is implemented in its own window. A layout of widgets occurs when several widgets are tiled side by side, or placed on top of each other (layered). For example, a window surface with text and scrollbars to control the positioning of that text is done by constructing a series of windows. The first window is the window that holds the presented text. The second and third are the windows that hold horizontal and vertical scrollbars.

Layering is done by defacto transparency. A series of widgets are placed one atop the other. For example, a text window can be laid on top of a background window.

The key to interface design is to think of a window as simply a building block for construction of the user interface.

Widgets are fundamental to the "look" of an interface. Because **widgets** uses the native widgets on the operating system it serves, the client program will pick up quite a bit of that look from just the use of the widgets.

There is more to an application than just the look of the widgets. There are layout conventions, actions, and other intangibles. The rule that applies to ANSI C portability is:

- ANSI C programs will be able to target a high percentage of simple applications just by use of its normal **widgets** components.
- ANSI C should be able to finish a high percentage of the work to create complex applications.

The typical cycle in designing an ANSI C application is to design an initial version that uses just **widgets** components, then finish the design with special layouts, actions and colors for a particular operating system that give it the "look and feel" of a native application.

11.2 Backgrond Colors and Placement

Widgets are placed by specifying their "bounding box" or rectangle. This is the rectangle that contains the widget. A widget can occupy all of the specified bounding box, or just a part of it. Typically, the operating system will try its best to format the widget to fit within the space provided.

The color scheme for widgets can vary. However, widgets are designed to be placed against a background color that is system dependent. This is available as a new system defined color, **pa_backcolor**. It can be selected by the standard color selection routines.

```
typedef enum { pa_black, pa_white, pa_red, pa_green, pa_blue,
               pa_cyan, pa_yellow, pa_magenta, pa_backcolor }

pa_fcolor(pa_backcolor);
pa_bcolor(pa_backcolor);
```

There are many ways to group widgets so that they blend into your background. The surface they appear on can be specifically colored using the background color, or you can create a child window with that color. Also, there is a background widget that can color the background for widgets automatically.

11.3 Sizes

Using widgets can go a long way towards getting the look and feel of a system in an independent way. The other important factor for achieving system independence is to control sizing of widgets. For each widget, with the exception of dialogs, there is a size routine. The size routine takes the particular features of that widget, such as face text or borders, and gives the best size for the widget, given the desired client area, contents, etc..

The size information must be considered against the particular widget to be created. Sizing is key to establishing the layout of widgets in a window, and key to producing a truly portable application.

11.4 Logical Widget Identifiers

A widget is created with a logical number, from 1 to n, where n is a positive integer. You specify the number you wish to create the widget under. The id of a widget cannot be the same as any other active widget, but widget logical identifiers can be reused by killing the widget first.

11.5 Killing, Selecting, Enabling and Getting Text to and from Widgets

A widget is killed by **pa_killwidget(f, id, e)**.

Some widgets can be selected, which changes their appearance to the select state. A widget is selected by **pa_selectwidget(f, id, e)**.

A selected widget can be a checkbox that is checked, a radio button that is pushed, or similar effect. The processing of selects, and keeping track of the state of widgets is up to you. However, this is a very flexible system. You can implement widgets that flip their state when clicked on, a series of widgets that are mutually exclusive, and many other combinations.

Similar to selection, widgets can be enabled or disabled by **pa_enablewidget(f, id, e).** Widgets are disabled by default. When a widget is disabled, it has the disabled appearance, such as greyed out. The widget will not give click events when it is disabled. Disabling a widget is typically used when it is not needed or not available in the current context. For example, a "next" button could be disabled when there is no next item to process.

Some widgets have text that can be set, read, or in some cases, both. The text in a widget is set by **pa_putwidgettext(f, id)**. The text in a widget can be read by **pa_getwidgettext(f, id)**.

Setting and getting the widget text is used with widgets that allow the user to modify the text, such as an edit box.

11.6 Resizing and repositioning a widget

To prevent the need to remove and replace widgets each time a window is resized, use **pa_sizwidget[g](f, id, x, y)**. To reposition the widget in the parent window, **pa_poswidget[g](f, id, x, y)** exists.

11.7 Types of widgets

Widgets come in three different types, controls, components and dialogs. Controls are widgets that the user can manipulate, and these widgets issue events to the program that owns them.

Components are display widgets whose only job is to form part of a display to the user. A group box, and a background are examples of components. Some components have active displays, such as the progress bar. Components never issue events.

Dialogs are fully autonomous windows that exist apart from the applications windows. They can be very complex inside, having a whole system of layout, widgets and other features. They resemble entirely separate programs.

Dialogs take a series of parameters when they are called, and deliver those same parameters back to the caller, with any modifications the user performs on the data.

11.8 Z ordering

widgets uses “implicit Z ordering” for layered widgets. In order for widgets to properly layer, the drawing Z order must be controlled. For example, a background must be drawn first, followed by whatever is on top of it, or the drawing of the background will wipe out what is on the surface of the background.

To enable this, **widgets** makes sure that widgets which are designed to be layered appear first in the drawing order. This means that controls are in front of components.

11.9 Controls

A button can be created with **pa_button[g](f, x1, xy1, x2, y2, s, id)**. The button is drawn in window **f**, in the specified rectangle (**x1**, **y1**) to (**x2**, **y2**) with a label text string **s**, and logical widget number **id**. The text string will be a single line of text with no control characters, and will be presented on the face of the button. The font style and size will be the same as other buttons in the operating systems user interface.

A button has different appearances when pressed vs. not pressed

When a button is pressed, it will typically change its appearance to indicate that. The button will send an event **pa_etbutton**. This event carries the id of the button that was asserted. Similarly, when a button is released, it changes appearance back from the pressed state.

Buttons can be any size, any height and width. The button face text does not get larger with the button, but remains centered. However, if the button too small, some of the face text will be clipped off. The size of the button can be determined before creating it by **pa_buttonsiz[g](f, s, w, h)**.

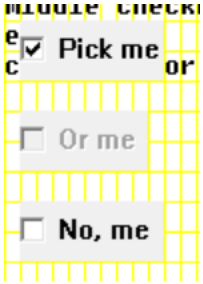
Buttons cannot be selected, but they can be disabled. The text in a button can be neither read nor written.

A disabled button.

A checkbox is created with **pa_checkbox[g](f, x1, y1, x2, y2, s, id)**. when hit, give a single event that indicates activation, **pa_etchkbox**. The event contains the identifier of the widget.

Checkbox sizing is found with **pa_checkboxsiz[g](f, s, w, h)**. Checkboxes are sized to minimum, but since they have no edges (like a button), there is typically no need to add space to them.

Checkboxes can be selected (checked). They can be enabled or disabled, and default to enabled. They cannot have their face text written or read.

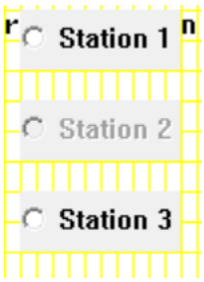


Checkboxes can be enabled, disabled or selected.

Radio buttons work identically to checkboxes, but have a different appearance. A radio button is created by **pa_radiobutton[g](f, x1, y1, x2, y2, s, id).** Radio buttons, when hit, give a single event that indicates activation, **pa_etradbut**. This event contains the id of the widget.

Radio button sizing is found with **pa_radiobuttonsizsiz[g](f, s, w, h)**. Radio buttons are sized to minimum, but since they have no edges (like a button), there is typically no need to add space to them.

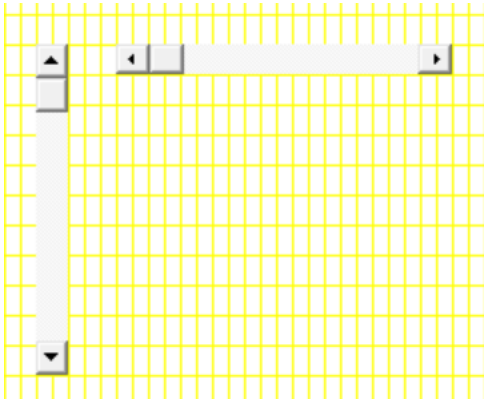
Radio buttons can be selected (checked). They can be enabled or disabled, and default to enabled. They cannot have their face text written or read.



Radio buttons can be enabled, disabled or selected.

Scrollbar widgets are free floating, and can appear anywhere in the window, not just the sides. In addition, the height and width of them can be controlled, instead of being fixed to the window size.

Scrollbars are placed vertically by **pa_scrollvert[g](f, x1, y1, x2, y2, id)**. Scrollbars are placed horizontally by **pa_scrollhoriz[g](f, x1, y1, x2, y2, id)**.



Horizontal and vertical scrollbars.

Scroll bars can be placed using any dimensions, but the width of a vertical scroll bar, and the height of a horizontal scroll bar usually has a standard size. These can be determined by **pa_scrollvertsiz[g](f, w, h)** and **pa_scrollhorizsiz[g](f, w, h).**

A user movement of a scrollbar is given by the event **pa_etsclpos**. This event does not move the scrollbar slider. This must be done by the program via **pa_scrollpos(f, id, p)**.

When a user positions the scroll bar directly, it will follow the users mouse movements. However, it will return to the original position unless the **pa_etsclpos** event is responded to and a **pa_scrollpos(f, id, p)** call is made.

Besides position events, scrollbars issue two types of button pushes, referred to as line and page button events. The line buttons are the arrow buttons at either side of the scroll bar. The page buttons are the space between the scrollbar slider and the line arrows. A press anywhere in this area generates a page button event.

The page button area may not appear at all if the slider is fully to one side of the scrollbar.

The page and line terminology for these buttons occurs because their most common use is to scroll documents. In this case, the line buttons would be used to move the document one line up or down. In the case of horizontal scrollbars, this would be one character left or right (despite the term "line" in the buttons name). The page refers to one screenful of text, in any direction. If the page up button is hit, for example, the document would move one screenful up.

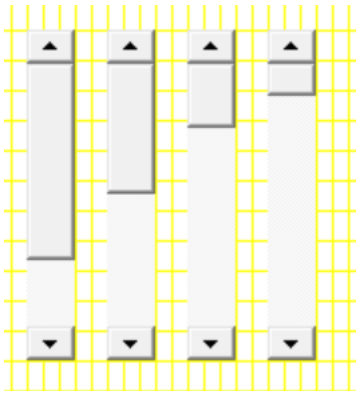
It's up to the program to implement the actions for line up/down and page up/down. In fact, these events can be used for any purpose in client programs.

Besides the position of the slider, its size can also be controlled by **pa_scrollsiz(f, id, s)**.

The standard use of the scrollbar size is to indicate what proportion of the document or display is contained within the onscreen display, vs. the total size of the document. For example, if the document has 50% of its total displayed, then the size of the scrollbar is set to %50, which is done by:

```
pa_scrollsiz(f, n, INT_MAX div 2);
```

Scroll bars cannot be selected, enabled or disabled, or have face text read or written.

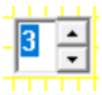


Scrollbars with sliders of various sizes.

A number can be selected in an edit box by **pa_numselbox[g](f, x1, y1, x2, y2, l, u, id)**.

The first number that appears in the number select box is by default the lower bound.

Number select boxes are an easy way to enter numbers from the user, and automatically restrict the input to numbers only. The numbers are entered in decimal, and cannot be negative. The edit box allows the number to be directly edited. Also, there are usually up and down buttons that allow the user to count the number up or down by one.



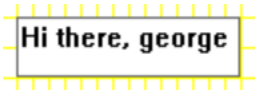
A number select box.

The size of a number select box is found by **pa_numselboxsiz[g](f, l, u, w, h)**.

Number select boxes cannot be selected, enabled or disabled, or have face text read or written.

A general string can be edited with **pa_editbox[g](f, x1, y1, x2, y2, id)**.

And empty edit box is placed, and the user has the ability to edit text into the box, with cursor movements, character delete, etc.



An edit box.

An edit box can be presented blank, or default text can be placed into the edit box. If the user presses enter to the box, it sends a **pa_etedtbox** event. However, the program can use any method to signal done, such as a button next to the edit control. The resulting text can then be retrieved from the edit box.

The size of an edit box is found by **pa_editboxsiz[g](f, s, w, h)**.

Edit boxes cannot be selected, enabled or disabled..

A list box is a series of items that can be selected. It is placed with **pa_listbox[g](f, x1, y1, x2, y2, sp, id)**, where **sp** is a list of strings to display.

The string list definition appears as:

```
/* string set for list box */
typedef struct pa_strrec* pa_strptr;
typedef struct pa_strrec {

    pa_strptr next; /* next entry in list */
    char*     str;  /* string */

} pa_strrec;
```

The string pointer is a list of strings, each string of which describes an entry in the list box.

When the user selects an item from the list box, the **pa_etlstbox** event is returned. This event gives the id of the widget, and the number of the select, from the top. The first item in the list will be 1, the second 2, etc.

The size of a list box is found by **pa_listboxsiz[g](f, sp, w, h)**.

A list box.

List boxes cannot be selected, enabled or disabled, or have face text read or written.

The same multiple string selection can be done in a different way by **pa_dropbox[g](f, x1, y1, x2, y2, sp, id)**.

A drop box only shows the whole list if the user selects it, otherwise only the currently selected entry is shown. The full list "drops down" from the selection box when the user selects it. A drop box takes less space than a list box when it is not selected. A drop box selection is signaled by the **pa_etdrpbox** event, which gives the widget id, and the number of the selection, from 1 to n.

A drop box, in both closed and open states.

The size of a drop box is found by **pa_dropboxsiz[g](f, sp, cw, ch, ow, oh)**. Because drop boxes have two appearances, one when open, and one when closed, both sizes are returned. The closed appearance gives the basic size of the widget, but the open size makes is possible to determine if the list will go beyond the edge of the window when open.

Drop boxes cannot be selected, enabled or disabled..

Very similar to a drop box, a drop/edit box allows selection from a list, but also allows the current selection string to be edited.

A drop/edit box is placed with **pa_dropeditbox[g](f, x1, y1, x2, y2, sp, id)**. When a selection is made from the drop/edit box, the **pa_etdrebox** event is sent, which includes the widget identifier. The selection data itself is a string, and must be retrieved with **pa_getwidgettext(f, id, s)**.

A drop/edit box in both closed and open states.

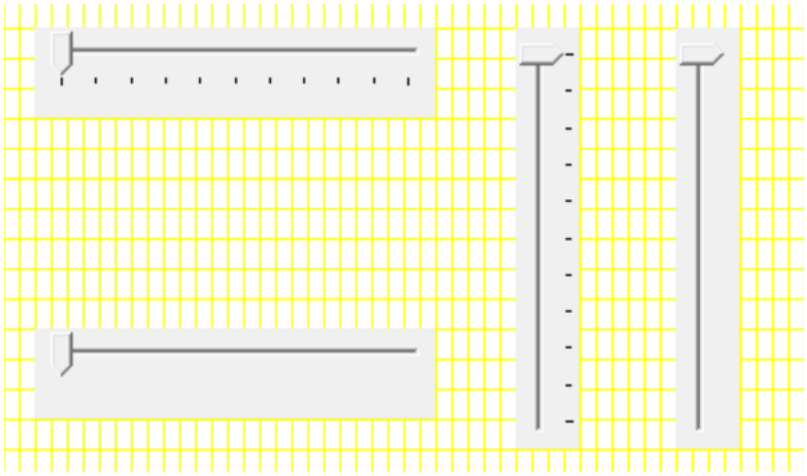
The size of a drop/edit box is found by **pa_dropeditboxsiz[g]**. Because drop/edit boxes have two appearances, one when open, and one when closed, both sizes are returned. The closed appearance gives the basic size of the widget, but the open size makes is possible to determine if the list will go beyond the edge of the window when open.

Drop/edit boxes cannot be selected, enabled or disabled.

Sliders are linear controls that can be placed either horizontally or vertically.

A vertical slider is placed with **pa_slidevert[g](f, x1, y1, x2, y2, m, id).** A horizontal slider can be placed by **pa_slidehoriz[g](f, x1, y1, x2, y2, m, id).** The m parameter gives the number of tick marks to place on the slider, which could be zero.

Sliders indicate changes in their position with the event **pa_etsldpos**. This gives the widget id, and a **INT_MAX** ratioed position of the slider, from 0 to **INT_MAX**. 0 is the top or leftmost position of the slider, and **INT_MAX** is the bottom or rightmost position of the slider.



Horizonal and vertical sliders, both with and without tick marks.

The size of a slider can be determined by **pa_scrollvertsiz[g](f, w, h)**.

Sliders cannot be selected, enabled or disabled, or have face text read or written.

Tab bars allow the user to select from a series of labeled tabs, usually to specify locations in document. A **tabbar** is a group box with tabs on one edges. The tabs can be placed on the top, bottom, left or right side of the included client area.

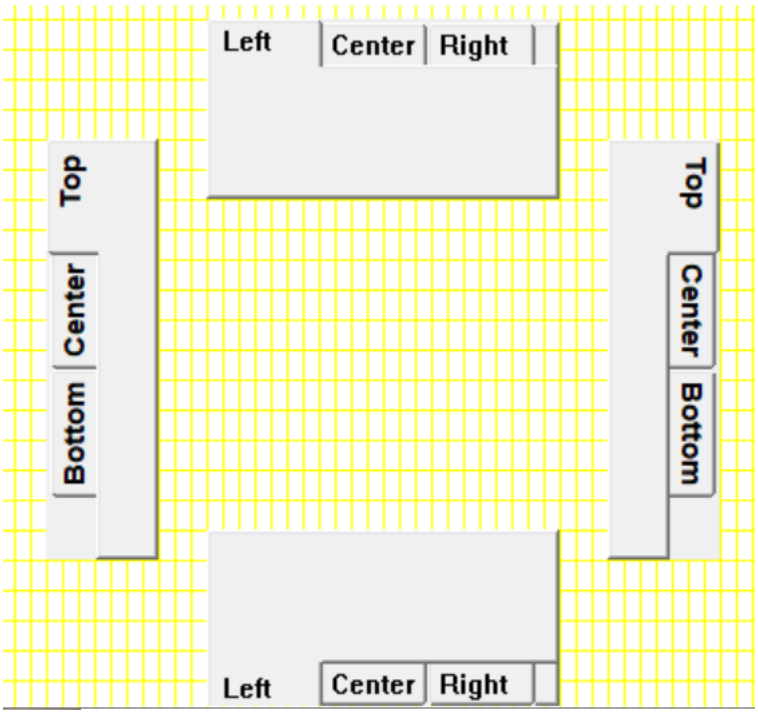
A tabbar is placed by **pa_tabbar[g](f, x1, y1, x2, y2, sp, to, id)**.

Tabbar selections are indicated by the event **pa_ettabbar**, which gives the widget id, and the tab number selection, from 1 to n, counting from the first string entry in the list.

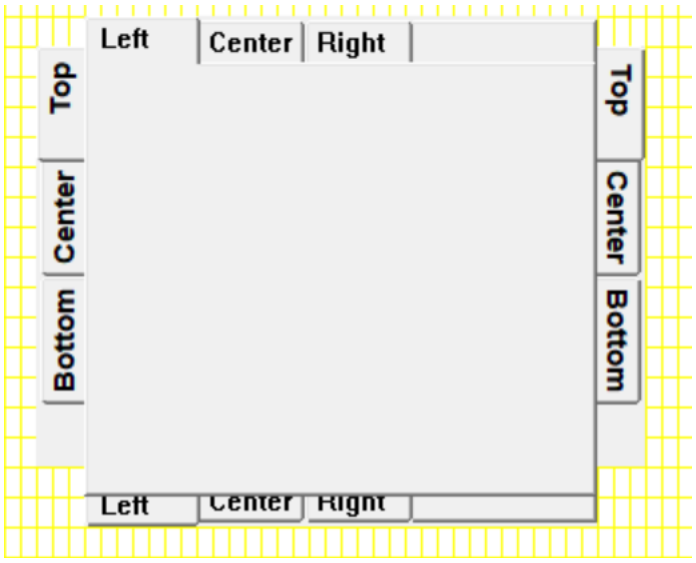
The size of a tabbar is found by **pa_tabbarsiz[g](f, to, cw, ch, w, h, ox, oy)**. A **tabbar** acts like a group box, and has a client area to place child windows or widgets. The required client size can be specified, and the sizing call returns the offset required to find the client location within the **tabbar**.

If the **tabbar** must fit into a fixed window size, the size of the resulting client for a **tabbar** can be found with **pa_tabbarclient[g](f, to, w, h, cw, ch, ox, oy)**. This returns the client width and height, and its offset from the origin of the **tabbar**.

Tab bars cannot be selected, enabled or disabled, or have face text read or written.



Separate left, top, right and bottom tab bars.



Overlaid tab bars that share a client area.

When a tab bar with tabs on more than one side is needed, several tab bars with different orientations can be overlaid.

```
/* orientation for tab bars */
typedef enum { pa_totop, pa_toright, pa_tobottom, pa_toleft }
pa_tabori;
```

11.10 Components

A background box is placed by **pa_background[g](f, x1, y1, x2, y2, id)**. A background box is designed to serve as the background to a series of controls, and it has the standard color for such backgrounds.

A background component. More useful than a rectangle because it draws itself.

Background boxes have no sizing, because there are no borders or other content. They are just a colored rectangle. Background boxes cannot be selected, enabled or disabled, or have face text read or written.

A group box is similar to a background box, but it has a label for the "group" of controls contained within it. It is placed by **pa_group[g](f, x1, y1, x2, y2, s, id)**.

A group box, which is a labeled background.

The size of a group box found by **pa_groupsiz[g](f, s, cw, ch, w, h, ox, oy)**. A group box has a client area to place child windows or widgets. The required client size can be specified, and the sizing call returns the offset required to find the client location within the group box.

Group boxes cannot be selected, enabled or disabled, or have face text read or written.

A progress bar is used to indicate the progress of a job completion, like installing software, saving a file, etc.

It is placed by **pa_progbar[g](f, x1, y1, x2, y2, id)**. The initial progress indication is zero when placed.

A progress bar.

The size of a progress bars can be determined by **pa_progbarsiz[g](f, w, h)**.

The position of the progress bar is set by **pa_progbarpos(f, id, pos)**.

Progress bars cannot be selected, enabled or disabled, or have face text read or written.

11.11 Dialogs

A dialog is a completely separate window which is preformatted with widgets. Dialogs introduce complex queries into a program, using the look of the native operating system,.

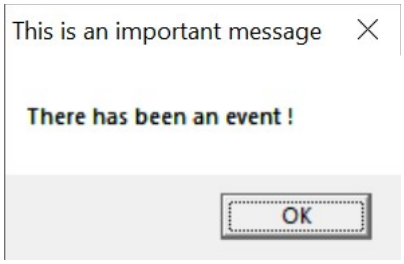
Dialogs display a property known as modality. Since the dialog is a separate window, it can be independent of the other windows created by the calling task, or the dialog can be forced to appear at the top of the applications stacking order.

widgets uses two types of modality. If the task that created the dialog also created other windows, the dialog is forced to the front of the other windows, and selection of the other task windows is disabled. This indicates to the user that only the dialog is currently being managed by the task.

If windows are created by different threads, then the dialog will not be modal vs. the other thread's windows. This reflects the fact that the windows outside the dialog can run while the dialog does.

An alert dialog is used to send errors or other important messages to the user. It has a window title, a message that constitutes the alert, and typically has an "ok" or "close" button for the user to indicate the user has seen it.

An alert is created by **pa_alert(title, msg).** The alert call will not return until the user has clicked the OK button for the alert.

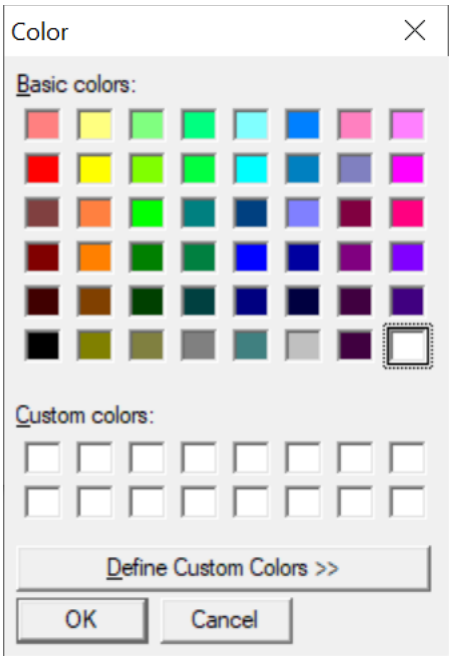


An alert box. One of the simplest and most useful widgets.

The query dialogs allow the user to select important information such as a file, a search string, or color or a font. They use a model called "flow through". The query may select several types of information, and will accept a default setting, allow the user to select a new setting, and return that. The flow through model means that several parameters are set up before the call, may or may not be modified by the query, then are returned to the caller. The calling thread stops until the dialog is completed by the user.

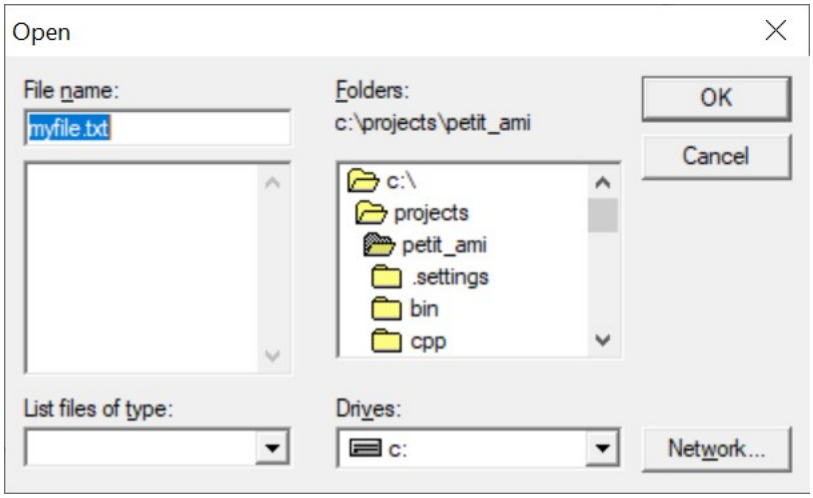
All of the parameters of a dialog may or may not be implemented in the actual system dialog. The flowthrough system allows for differences in systems. Since the variables are preinitalized with the defaults, if the dialog does not implement a particular parameter, the value will be left unchanged.

A color can be chosen by **pa_querycolor(r, g, b).** The default color is set before the call, and the possibly changed color is returned by the call.



A color query dialog.

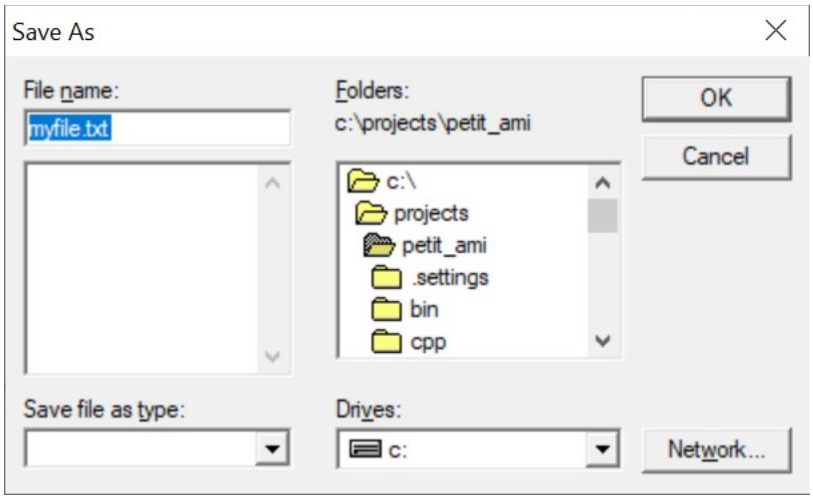
A file to open name is selected by **pa_queryopen(s, sl)**.



A file open dialog.

The default filename is passed in **s**, and the resulting filename string returned in **s** with maximum length **sl**. If the dialog is canceled instead of completed by the user, the string returned is empty. This means to not proceed with the open operation.

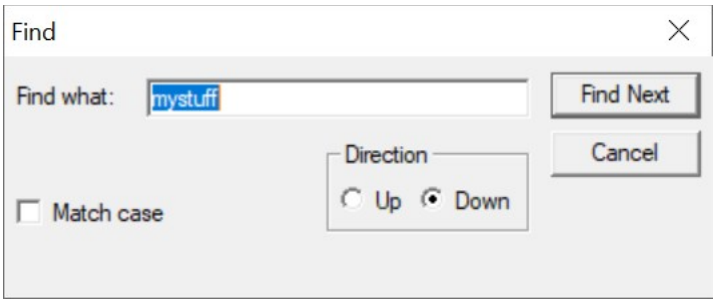
A file to save name is selected by **pa_querysave(s, sl)**.



A file save dialog.

The default filename is passed in **s**, and the resulting filename string returned in **s** with maximum length **sl**. If the dialog is canceled instead of completed by the user, the string returned is empty. This means to not proceed with the save operation.

A string to search for is selected by **pa_queryfind(s, sl, opt)**.



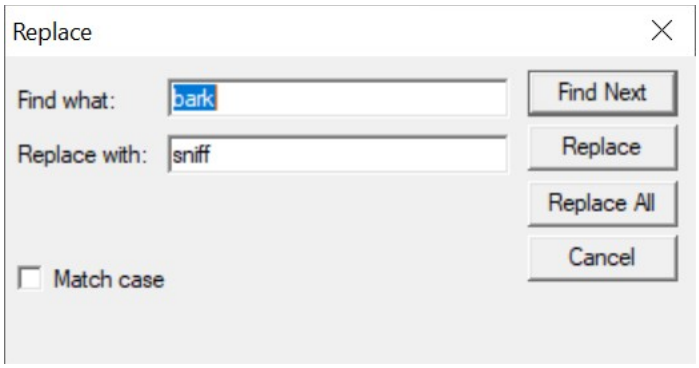
A find dialog.

The option flags are given by a set of flags:

```
/* settable items in find query */
typedef enum { pa_qfncase, pa_qfnup, pa_qfnre } pa_qfnopt;
typedef int pa_qfnopts;
```

The default search string is passed in s, and the resulting search string returned in **s** with maximum length **sl**. If the dialog is canceled instead of completed by the user, the string returned is empty. This means to not proceed with the search operation.

A string to search for and replace is selected by **pa_queryfindrep(s, sl, r, rl, opt)**.



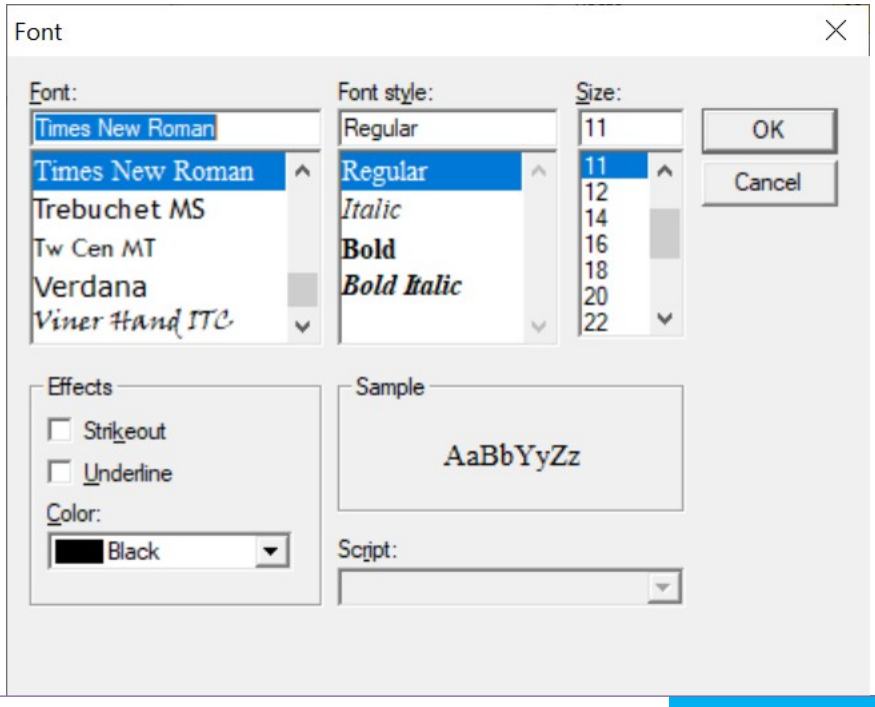
A find/replace dialog.

The option flags are given by a set of flags:

```
/* settable items in replace query */
typedef enum { pa_qfrcase, pa_qfrup, pa_qfrre, pa_qfrfind,
pa_qfrallfil, pa_qfralllin } pa_qfropt;
typedef int pa_qfropts;
```

The default search string and replacement strings are passed in s and r, and the resulting search string is returned in **s** with maximum length **sl**, and the resulting replacement string is returned in **r** with maximum length **rl**. If the dialog is canceled instead of completed by the user, both strings are returned empty. This means to not proceed with the search/replace operation.

Fonts are selected by **pa_queryfont(f, fc, s, fr, fg, fb, br, bg, bb, effect)**.



A font query dialog.

The font effects are declared as:

```
/* effects in font query */
typedef enum { pa_qfteblink, pa_qftereverse, pa_qfteunderline,
pa_qftesuperscript,
                  pa_qftesubscript, pa_qfteitalic, pa_qftebold,
pa_qftestrikeout,
                  pa_qftestandout, pa_qftecondensed, pa_qfteextended,
pa_qftexlight,
                  pa_qftelight, pa_qftexbold, pa_qftehollow,
pa_qfteraised} pa_qfteffect;
typedef int pa_qfteffects;
```

11.12 Events

The definition of an event record is upward compatible with previous event record declarations from **terminal**, **graphics** and **windows**.

```
/* events */
typedef enum {
    pa_etchar,      /* ANSI character returned */
    pa_etup,        /* cursor up one line */
    pa_etdown,      /* down one line */
    pa_etleft,      /* left one character */
    pa_etright,     /* right one character */
    pa_etleftw,     /* left one word */
    pa_etrightw,    /* right one word */
    pa_ethome,      /* home of document */
    pa_ethomes,     /* home of screen */
    pa_ethomel,     /* home of line */
    pa_etend,       /* end of document */
    pa_etends,      /* end of screen */
    pa_etendl,      /* end of line */
    pa_etscrl,      /* scroll left one character */
    pa_etscrr,      /* scroll right one character */
    pa_etscru,      /* scroll up one line */
    pa_etscrd,      /* scroll down one line */
    pa_etpagd,      /* page down */
    pa_etpagu,      /* page up */
    pa_ettab,       /* tab */
    pa_etenter,     /* enter line */
    pa_etinsert,    /* insert block */
    pa_etinsertl,   /* insert line */
    pa_etinsertt,   /* insert toggle */
    pa_etdel,       /* delete block */
    pa_etdell,      /* delete line */
    pa_etdelcf,     /* delete character forward */
    pa_etdelcb,     /* delete character backward */
    pa_etcopy,      /* copy block */
    pa_etcopyl,     /* copy line */
    pa_etcan,       /* cancel current operation */
    pa_etstop,      /* stop current operation */
    pa_etcont,      /* continue current operation */
    pa_etprint,     /* print document */
    pa_etprintb,    /* print block */
    pa_etprints,    /* print screen */
    pa_etfun,       /* function key */
    pa_etmenu,      /* display menu */
    pa_etmouba,     /* mouse button assertion */
    pa_etmoubd,     /* mouse button deassertion */
    pa_etmoumov,    /* mouse move */
    pa_ettim,       /* timer matures */
    pa_etjoyba,     /* joystick button assertion */
```

```
    pa_etjoybd,    /* joystick button deassertion */
    pa_etjoymov,   /* joystick move */
    pa_etresize,   /* window was resized */
    pa_etterm,     /* terminate program */
    pa_etframe,    /* frame sync */
    pa_etmoumovg,  /* mouse move graphical */
    pa_etredraw,   /* window redraw */
    pa_etmin,      /* window minimized */
    pa_etmax,      /* window maximized */
    pa_etnorm,     /* window normalized */
    pa_etmenus,    /* menu item selected */
    pa_etbutton,   /* button assert */
    pa_etchkbox,   /* checkbox click */
    pa_etradbut,   /* radio button click */
    pa_etsclull,   /* scroll up/left line */
    pa_etscldrl,   /* scroll down/right line */
    pa_etsclulp,   /* scroll up/left page */
    pa_etscldrp,   /* scroll down/right page */
    pa_etsclpos,   /* scroll bar position */
    pa_etedtbox,   /* edit box signals done */
    pa_etnumbox,   /* number select box signals done */
    pa_etlstbox,   /* list box selection */
    pa_etdrpbox,   /* drop box selection */
    pa_etdrebox,   /* drop edit box signals done */
    pa_etsldpos,   /* slider position */
    pa_ettabbar    /* tab bar select */
} pa_evtcod;
/* event record */
typedef struct {

    /* identifier of window for event */ int winid;
    /* event type */                     pa_evtcod etype;
    /* event was handled */              int handled;
    union {

        /* these events require parameter data */

        /** etchar: ANSI character returned */  char echar;
        /** ettim: timer handle that matured */ int timnum;
        /** etmoumov: */
        struct {

            /** mouse number */   int mmoun;
            /** mouse movement */ int moupx, moupy;

        };
        /* etmouba */
        struct {
```

```
    /** mouse handle */  int amoun;
    /** button number */ int amoubn;

};
/* etmoubd */
struct {

    /** mouse handle */  int dmoun;
    /** button number */ int dmoubn;

};
/* pa_etjoyba */
struct {

    /** joystick number */ int ajoyn;
    /** button number */   int ajoybn;

};
/* pa_etjoybd */
struct {

    /** joystick number */ int djoyn;
    /** button number */   int djoybn;

};
/* pa_etjoymov */
struct {

    /** joystick number */      int mjoyn;
    /** joystick coordinates */ int joypx, joypy, joypz;

};
/* pa_etfun */
/** function key */ int fkey;
/** etmoumovg: */
struct {

    /** mouse number */   int mmoung;
    /** mouse movement */ int moupxg, moupyg;

};
/** etredraw */
struct {

    /** bounding rectangle */
    int rsx, rsy, rex, rey;

};
/* pa_etmenus */
```

```
int menuid; /* menu item selected */
/* pa_etbutton */
int butid; /* button id */
/* pa_etchkbox */
int ckbxid; /* checkbox */
/* pa_etradbut */
int radbid; /* radio button */
/* pa_etsclull */
int sclulid; /* scroll up/left line */
/* pa_etscldrl */
int scldlid; /* scroll down/right line */
/* pa_etsclulp */
int sclupid; /* scroll up/left page */
/* pa_etscldrp */
int scldpid; /* scroll down/right page */
/* pa_etsclpos */
struct {

    int sclpid; /* scroll bar */
    int sclpos; /* scroll bar position */

};
/* pa_etedtbox */
int edtbid; /* edit box complete */
/* pa_etnumbox,   /* number select box signals done */
struct {

    int numbid; /* num sel box select */
    int numbsl; /* num select value */

};
/* pa_etlstbox */
struct {

    int lstbid; /* list box select */
    int lstbsl; /* list box select number */

};
/* pa_etdrpbox */
struct {

    int drpbid; /* drop box select */
    int drpbsl; /* drop box select */

};
/* pa_etdrebox */
int drebid; /* drop edit box select */
/* pa_etsldpos */
struct {
```

```
            int sldpid; /* slider position */
            int sldpos; /* slider position */

        };
        /* pa_ettabbar */
        struct {

            int tabid;  /* tab bar */
            int tabsel; /* tab select */

        };

    };

} pa_evtrec, *pa_evtptr;
```

11.13 Procedures and functions in widgets

void pa_killwidget(FILE* f, int id);

The widget within window **f** with the logical identifier **id** is removed from the system. It will be erased from the screen, and contents under it will be restored as required.

void pa_selectwidget(FILE* f, int id, int e);

The widget within window **f** by the logical identifier **id** will enter the select state if **e** is true, otherwise the select state is removed. The exact effect of the select state depends on the widget. See the individual widget to be selected for more information. It is an error to select a widget that has no selectability.

Selection is typically used to indicate that the widget is "on", by changing its face appearance. For example, it can be checked, pressed or a similar visual state change.

void pa_enablewidget(FILE* f, int id, int e);

The widget within window **f** by the logical identifier **id** will enter the enable state if **e** is true, otherwise the disable state is entered. The exact effect of the enable or disable state depends on the widget. See the individual widget to be selected for more information. It is an error to enable or disable a widget that has no such capability. The default state for all widgets is to be enabled. A widget that is disabled will stop sending events.

Disabling a widget is typically used to mark it as unusable or invalid in the current context. An example would be a "next" button where no next page or item exists. The standard method used to indicate disabled widgets is to give them a "greyed out" appearance, but the actual effect may depend on the operating system.

void pa_getwidgettext(FILE* f, int id, char* s, int sl);

Retrieves the text contained by the widget with the logical identifier **id** within window **f**, and returns the text in string **s**, whose buffer length is **sl**. It depends on the widget as to if it has text that can be read. It is an error to get text from a widget that has no such capability.

Widgets can typically have their text read if the widget provides the user with the ability to modify or edit text. In this case, retrieving the text is required to obtain the new text.

void pa_putwidgettext(FILE* f, int id, char* s);

Places text in the widget with the logical identifier **id** within window **f** from the string **s**. It depends on the widget as to if it can accept text placed in this manner. It is an error to place text in a widget that has no such capability.

A widget will have the ability to place text if it can edit text by the user. Placing text in the widget can be used to initialize the contents of such a widget, or as part of the overall interaction with the user.

void pa_sizwidget[g](FILE* f, int id, int x, int y);

Resize an existing widget. The widget with the logical identifier **id** is resized to be the size in **x** and **y**, in the window **f**.

If the graphical version is used, the size is in pixels, otherwise, the size is in characters.

void pa_poswidget[g](FILE* f, int id, int x, int y);

Reposition an existing widget. The widget with the logical identifier **id** is repositioned to be at the position **x** and **y**, in the parent window of the window f.

If the graphical version is used, the size is in pixels, otherwise, the size is in characters.

void pa_buttonsiz[g](FILE* f, char* s, int* w, int* h);

Finds the minimum size of a button within window **f**, with face text string **s**. The width to use is returned in **w**, and the height in **h**. Button sizing returns the minimum size in terms of the minimum amount of space needed to contain the face text, in the labeling font, and the border of the button itself. You will want to use the size as a guide to button sizing, and not for the actual size of the button. A good rule of thumb is to add %25 of the minimum height of the button to the actual height and width of the button to give the user sufficient area to click on the button. In addition, buttons should be "justified" when they appear in groups to appear as the same width. This is done by calculating a maximum size for a group of related buttons, then using the same width and height for all of them.

void pa_button[g](FILE* f, int x1, int y1, int x2, int y2, char* s, int id);

Places a button within window **f** in the bounding box formed by **x1**, **y1**, **x2**, **y2**, with the face text from the string **s**. The logical identifier is specified in **id**, which must be an integer from 1 to n, which is not currently in use in any other widget. The button drawn is not guaranteed to completely fill the bounding box. The button should be placed against a background that is colored with the standard background color. The button will be placed at the front of the window stacking order, and should be placed after any other widgets or child windows that the button is to appear in front of.

When the button is pressed, it will send an **etbutton** event, which contains the logical id of the button that was pressed. It is up to the system whether the event occurs when the button is depressed or released.

Buttons cannot be selected, or have their face text changed or read. Buttons can be enabled and disabled. If the button is disabled, it will not send **etbutton** events when pressed.

void pa_checkboxsiz[g](FILE* f, char* s, int* w, int* h);

Finds the minimum size of a checkbox within window **f**, with face text string **s**. The width to use is returned in **w**, and the height in **h**. Checkbox sizing returns the minimum size in terms of the minimum amount of space needed to contain the face text, in the labeling font, and the border of the checkbox itself, if it exists. You will want to use the size as a guide to checkbox sizing, and not for the actual size of the checkbox. A good rule of thumb is to add %25 of the minimum height of the checkbox to the actual height and width of the checkbox to give the user sufficient area to click on the checkbox.

void pa_checkbox[g](FILE* f, int x1, int y1, int x2, int y2, char* s, int id);

Places a checkbox within window **f** in the bounding box formed by **x1**, **y1**, **x2**, **y2**, with the face text from the string **s**. The logical identifier is specified in **id**, which must be an integer from 1 to n, which is not currently in use in any other widget. The checkbox drawn is not guaranteed to completely fill the bounding box. The checkbox should be placed against a background that is colored with the standard background color. The checkbox will be placed at the front of the window stacking order, and should be placed after any other widgets or child windows that the checkbox is to appear in front of.

When the checkbox is clicked, it will send an **etchkbox** event, which contains the logical id of the checkbox that was pressed. It is up to the system whether the event occurs when the checkbox is depressed or released.

Checkboxes cannot have their face text changed or read. A checkbox can be selected or deselected, and can be enabled and disabled. If the checkbox is selected, it will appear with a selected face, which is typically a checkmark in a box. If the checkbox is disabled, it will not send **etchkbox** events when pressed.

A checkbox only changes its appearance in response to a select, and does not keep a state that can be read by the program. It's up to the program to keep track of the state of the checkbox, and how to handle it. In particular, if the **checkbox** is pressed, it is up to the program to change its select status, otherwise the press will have no effect. The program can implement many different effects for checkboxes. The **checkbox** can toggle, or it can be one of a series of mutually exclusive selections.

void pa_radiobuttonsiz[g](FILE* f, char* s, int* w, int* h);

Finds the minimum size of a radio button within window **f**, with face text string **s**. The width to use is returned in **w**, and the height in **h**. Radio button sizing returns the minimum size in terms of the minimum amount of space needed to contain the face text, in the labeling font, and the border of the radio button, if it exists. You will want to use the size as a guide to radio button sizing, and not for the actual size of the checkbox. A good rule of thumb is to add %25 of the minimum height of the radio button to the actual height and width of the radio button to give the user sufficient area to click on the radio button.

void pa_radiobutton[g](FILE* f, int x1, int y1, int x2, int y2, char* s, int id);

Places a radio button within window **f** in the bounding box formed by **x1**, **y1**, **x2**, **y2**, with the face text from the string **s**. The logical identifier is specified in **id**, which must be an integer from 1 to n, which is not currently in use in any other widget. The radio button drawn is not guaranteed to completely fill the bounding box. The radio button should be placed against a background that is colored with the standard background color. The radio button will be placed at the front of the window stacking order, and should be placed after any other widgets or child windows that the radio button is to appear in front of.

When the radio button is pressed, it will send an **etradbut** event, which contains the logical id of the radio button that was pressed. It is up to the system whether the event occurs when the radio button is depressed or released.

Radio buttons cannot have their face text changed or read. A radio button can be selected or deselected, and can be enabled and disabled. If the radio button is selected, it will appear with a selected face, which is typically a blacked out radio button. If the radio button is disabled, it will not sent **etradbut** events when pressed.

A radio button only changes its appearance in response to a select, and does not keep a state that can be read by the program. It's up to the program to keep track of the state of the radio button, and how to handle it. In particular, if the checkbox is pressed, it is up to the program to change its select status, otherwise the press will have no effect. The program can implement many different effects for radio buttons. The radio button can toggle, or it can be one of a series of mutually exclusive selections.

void pa_groupsiz[g](FILE* f, char* s, int cw, int ch, int* w, int* h, int* ox, int* oy);

Finds the required size of a group box in window **f**, with the face text given in string **s**, and the client area width **cw**, and client area height **ch**. The required width is returned in **w**, the height in **h**, and the offset to the client area in **ox** and **oy**. A group box consists of a border area, a label, and an internal client area. Group boxes are designed to be layered components. They contain other widgets, and provide a background for them. When a group size is found, the minimum size is found as what will contain all of the border, face text and the requested client area. The program will know where to place its widgets in the client area by the client offset, which is given as a difference between the bounding box origin, and the origin of the client rectangle.

void pa_group[g](FILE* f, int x1, int y1, int x2, int y2, char* s, int id);

Creates a group box in window **f** within the bounding rectangle **x1**, **y1**, **x2**, **y2**, the face text given in string **s**, and with the logical identifier **id**. Group boxes are containers for other widgets, and consist of a border area, the face text, and an internal client area where other widgets are to be placed.

The entire client area of the group will have the standard background color, and widgets can be placed into the client in any arrangement or number. The program should be sure to create the client area widgets after the group is created, so that they will appear in front of the group in stacking order.

The location of the client area within a group box can be found with the group sizing call.

void pa_background[g](FILE* f, int x1, int y1, int x2, int y2, int id);

Creates a background box in the window **f**, with the bounding rectangle **x1**, **y1**, **x2**, **y2**, and the logical widget identifier **id**. A background is simply a rectangle with the standard background color. It is more convenient than simply painting a rectangle on the window with the background color because it handles its own redraws. Because a background box has no borders, widgets can be placed within it anywhere, and in any number. Any widgets to be placed within the group should be created after the group box is created, so that that they are on top of the group box in stacking order.

void pa_scrollvertsiz[g](FILE* f, int* w, int* h);

Finds the size for a vertical scroll bar in window **f**. Returns the width in **w**, and the height in **h**. Scrollbars typically can be sized to any size, and the width of a vertical scroll bar is a suggested width designed to match others used in the same system. The height of a vertical scrollbar is simply a suggestion, and can be ignored.

If a scrollbar cannot be arbitrarily sized, then the width and height will reflect the dimensions of a fixed scrollbar.

void pa_scrollvert[g](FILE* f, int x1, int y1, int x2, int y2, int id);

Creates a vertical scrollbar in window **f**, with bounding rectangle **x1**, **y1**, **x2**, **y2**, and logical widget identifier **id**. If possible, the scrollbar is made to fill the width and height requested. If this is not possible, then the largest scrollbar is created that fits within the bounding rectangle. There is no guarantee that the scrollbar will completely fill the rectangle. The area under the scrollbar should be drawn with the standard background color.

The scrollbar will generate several events when clicked. The **etsclull** event indicates the line up button of the scrollbar was pressed. The **etscldrl** event indicates the line down button of the scrollbar was pressed. The **etsclulp** event indicates the page up section of the scrollbar was pressed. The **etscldrp** event indicates the page down section of the scrollbar was pressed. It is system dependent as to whether the buttons generate their events on a button press or a button release.

The **etsclpos** event gives the position of the top of the slider after the user moves it. The position is returned as a ratioed **INT_MAX** number, where 0 means the slider is at the top, and **INT_MAX** means the slider is at the bottom. The number is affected by the size of the slider. If, for example, the slider occupies %50 of the scrollbar, then only the positions 0 to **INT_MAX div** 2 will be generated. It is undefined as to exactly when **etsclpos** events occur. They may only be generated when the slider is moved and then released, or they may be generated continuously as the slider is moved. If the generation is continuous, then the movements are usually subject to "rate limiting" to keep them from generating events too fast to handle.

The scrollbar will not change position on its own. The scrollbar will generate events, and it is up to the program to use those to set the scrollbar slider position, and to take action on them, such as move the screen data up or down. The page up/down and line up/down terminology is suggestive of the use of these events, but it is up to the program exactly how to use or implement these functions. Typically, these are used to move the displayed area of a document one line up or down, and one page or screenful up or down.

The size of the scrollbar slider is set by default to small, but convenient for the user to press and manipulate. This can be left alone for programs that don't require sized scrollbar sliders. The size of the slider is set by **pa_scrollsiz[g]()**. Typically, it is used to set the ratio of onscreen data shown to the entire document or other data available. For example, if %50 of the document is being displayed, then the slider should occupy %50 of the scrollbar.

void pa_scrollhorizsiz[g](FILE* f, int* w, int* h);

Finds the size for a horizontal scroll bar in window **f**. Returns the width in **w**, and the height in **h**. Scrollbars typically can be sized to any size, and the height of a horizontal scroll bar is a suggested width designed to match others used in the same system. The width of a horizontal scrollbar is simply a suggestion, and can be ignored.

If a scrollbar cannot be arbitrarily sized, then the width and height will reflect the dimensions of a fixed scrollbar.

void pa_scrollhoriz[g](FILE* f, int x1, int y1, int x2, int y2, int id);

Creates a horizontal scrollbar in window **f**, with bounding rectangle **x1**, **y1**, **x2**, **y2**, and logical widget identifier **id**. If possible, the scrollbar is made to fill the width and height requested. If this is not possible, then the largest scrollbar is created that fits within the bounding rectangle. There is no guarantee that the scrollbar will completely fill the rectangle. The area under the scrollbar should be drawn with the standard background color.

The scrollbar will generate several events when clicked. The **etsclull** event indicates the line left button of the scrollbar was pressed. The **etscldrl** event indicates the line right button of the scrollbar was pressed. The **etsclulp** event indicates the page left section of the scrollbar was pressed. The **etscldrp** event indicates the page right section of the scrollbar was pressed. It is system dependent as to whether the buttons generate their events on a button press or a button release.

The **etsclpos** event gives the position of the left of the slider after the user moves it. The position is returned as a ratioed **INT_MAX** number, where 0 means the slider is at the left, and **INT_MAX** means the slider is at the right. The number is affected by the size of the slider. If, for example, the slider occupies %50 of the scrollbar, then only the positions 0 to **INT_MAX div** 2 will be generated. It is undefined as to exactly when **etsclpos** events occur. They may only be generated when the slider is moved and then released, or they may be generated continuously as the slider is moved. If the generation is continuous, then the movements are usually subject to "rate limiting" to keep them from generating events too fast to handle.

The scrollbar will not change position on its own. The scrollbar will generate events, and it is up to the program to use those to set the scrollbar slider position, and to take action on them, such as move the screen data left or right. The page left/right and line left/right terminology is suggestive of the use of these events, but it is up to the program exactly how to use or implement these functions. Typically, these are used to move the displayed area of a document one character left or right, and one page or screenful left or right.

The size of the scrollbar slider is set by default to small, but convenient for the user to press and manipulate. This can be left alone for programs that don't require sized scrollbar sliders. The size of the slider is set by **pa_scrollsiz[g]()**. Typically, it is used to set the ratio of onscreen data shown to the entire document or other data available. For example, if %50 of the document is being displayed, then the slider should occupy %50 of the scrollbar.

void pa_scrollpos(FILE* f, int id, int r);

Sets the scrollbar slider position for window **f**, scrollbar identifier **id**, to position **p**. The position is in ratioed **INT_MAX** format. That is, 0 means to set the position to the top or left, and **INT_MAX** means bottom or right. The position is affected by the size of the scrollbar slider. For example, if the slider occupies %50 of the scrollbar, then the range of positions would only be from 0 to **INT_MAX** div 2. If the position given is beyond the maximum position possible, then the slider is set to the maximum travel position, and no error occurs. It is an error if the position is negative.

The program must specifically set the position of the scrollbar. The user moving the scrollbar slider may temporarily move the slider while it is being moved, but this will not remain in position after the user releases it. The program must set the scrollbar position in response to the event.

void pa_scrollsiz(FILE* f, int id, int r);

Sets the size of the scrollbar slider in window **f**, logical identifier **id**, to the size **s**. The size of the scrollbar slider is an **INT_MAX** ratio, with 0 meaning infinitely small, and **INT_MAX** meaning that it occupies the entire scrollbar. In practice, there is a practical limit to how small the slider can be. If the slider is set too small, it will be set to the minimum size, and no error will occur. If the size set is negative, then an error will result.

The size of the scrollbar is set to a reasonable default if it is never specifically set. This is typically a fairly small size that is still easy to press and manipulate by the user. This allows the scrollbar to be used when slider sizing is not supported by the program.

The meaning of the scrollbar slider size is up to the program. However, it is typically used to indicate how much of the data is onscreen. For example, if a document has %50 of its content currently displayed, then the slider would be set to %50 of the scrollbar.

void pa_numselboxsiz[g](FILE* f, int l, int u, int* w, int* h);

Finds the width and height of a number select box for window **f**, with lower number limit **l** and upper number limit **u**. The width required is returned in **w**, and the height in **h**. The minimum width and height is determined by the maximum length of the number to be displayed, with borders and up/down arrows considered. This can be used without adding extra space.

void pa_numselbox[g](FILE* f, int x1, int y1, int x2, int y2, int l, int u, int id);

Creates a number select box for window **f**, in the rectangle **x1**, **y1**, **x2**, **y2**, with lower number limit **l**, and upper number limit **u**. The default number appearing in the box is set to the lower limit. The number select box allows the user to edit the number, or use up/down arrow controls to select the number. Any digits typed into the edit section are limited to the digits 0-9, and negative numbers are not allowed. When the user presses enter to the number edit box, an event, **etnumbox** will be sent, which includes the number selected.

void pa_editboxsiz[g](FILE* f, char* s, int* w, int* h);

Finds the size of an edit box for window **f**, with face text string **s**. The width is returned in **w**, and the height in **h**. The string passed is a dummy, and will not be used for any purpose other than as a reference to determine the width of the required edit box. The string should contain text that is representative of the string to be edited. This could be the string that you plan to place in the edit box as its default, or it could be the worst case contents of the edit box. For example, if 8 characters is the planned edit width, the string "WWWWWWWW" (8 "W" characters) would be the largest width of edit possible. The edit box is sized to be the minimum appropriate, and can be used without extra added space.

void pa_editbox[g](FILE* f, int x1, int y1, int x2, int y2, int id);

Creates an edit box for window **f**, in rectangle **x1**, **y1**, **x2**, **y2**, with logical identifier **id**. Edit boxes can be used to allow the user to enter any text. The text within an edit box can be set by **pa_putwidgettext()**, and retrieved by **pa_getwidgettext()**. This can occur at any time. When the user presses enter in the edit box, it sends an **pa_etedtbox** event. The program can then retrieve the text from the edit box.

void pa_progbarsiz[g](FILE* f, int* w, int* h);

Finds the size of a progress bar for window **f**. The width is returned in **w**, and the height in **h**. For systems that can size progress bars arbitrarily, the height is returned as the size that matches others used in the system. The width is a suggestion, and can be ignored.

void pa_progbar[g](FILE* f, int x1, int y1, int x2, int y2, int id);

Creates a progress bar in window **f**, in rectangle **x1**, **y1**, **x2**, **y2**, with logical identifier **id**. The progress bar starts by default at 0, and is entirely operated by the program with **pa_progbarpos()** calls.

void pa_progbarpos(FILE* f, int id, int pos);

Sets the progress bar in window **f**, with logical identifier **id**, to the position **pos**. The position is a ratioed **INT_MAX** number, from 0 to **INT_MAX**. 0 indicates "no progress", and **INT_MAX** indicates "complete". Because of rounding, it is recommended that the program specifically set **INT_MAX** at completion, instead of using a formula.

void pa_listboxsiz[g](FILE* f, pa_strptr sp, int* w, int* h);

Finds the required size of a listbox for window **f**, with string list **sp**. The required width is returned in **w**, and the required height is returned in **h**. A listbox is sized such that all of the strings in the string list can be presented in it, with borders added. No extra space is required.

void pa_listbox[g](FILE* f, int x1, int y1, int x2, int y2, pa_strptr sp, int id);

Creates a listbox for window **f**, in rectangle **x1**, **y1**, **x2**, **y2**, with string list **sp**, and logical identifier **id**. A listbox contains a series of strings that can be selected by the user. When the user clicks a string, the event **pa_etlstbox** will be sent, which contains the number of the selected string in list order. For example, the first string in the list would be 1, and second string in the list 2, etc. If there is not enough room in the height of the listbox for all strings in the list to be presented, then the widget will use a compression method to fit the available space. This is typically done by allowing the user to scroll through the selections. If there is not enough width for the strings in the list, they are typically clipped at the right.

void pa_dropboxsiz[g](FILE* f, pa_strptr sp, int* cw, int* ch, int* ow, int* oh);

Finds the size of a drop box for window **f**, with string list sp. The closed width is returned in **cw**, and the closed height in **h**. The open width is returned in **ow**, and the height in **oh**. Drop boxes are used to display a list of selections as in a listbox, but they occupy less space than a listbox. Dropboxes have two bounding rectangles, one is its dimensions when closed, and another when the user drops it down, or opens it. Both sizes are returned. This allows layout planning for both modes of the widget. Generally, the closed size is used to plan placement, then the open size is used to check if the widget will extend past the edges of the window when open.

void pa_dropbox[g](FILE* f, int x1, int y1, int x2, int y2, pa_strptr sp, int id);

Creates a dropbox in the window **f**, within rectangle **x1**, **y1**, **x2**, **y2**, for string list **sp**, with logical identifier **id**. The bounding rectangle for a drop box specifies its open mode, where the user has selected and dropped down the list of items. If the size specified is greater than or equal to the open size, as determined by **pa_dropboxsiz()**, then the entire dropbox will be presented. If the size is between the closed size and the open size, the system will attempt to work around the fact that the entire list cannot be dropped down. This is typically done by allowing the user to scroll though the list. If the size is less than the closed size, the dropbox will be clipped.

When a string within the drop box is selected, it will send an **pa_etdrpbox** event. It contains the sequential number of the string that was selected. For example, the first string sends 1, the second in the list sends 2, etc.

void pa_dropeditboxsiz[g](FILE* f, pa_strptr sp, int* cw, int* ch, int* ow, int* oh);

Finds the size of a drop edit box for window **f**, with string list sp. The closed width is returned in **cw**, and the closed height in **ch**. The open width is returned in **ow**, and the height in **oh**. Drop edit boxes are used to display a list of selections, and acts as a combination of a list and edit box, but they occupy less space than a listbox. Drop edit boxes have two bounding rectangles, one is its dimensions when closed, and another when the user drops it down, or opens it. Both sizes are returned. This allows layout planning for both modes of the widget. Generally, the closed size is used to plan placement, then the open size is used to check if the widget will extend past the edges of the window when open.

void pa_dropeditbox[g](FILE* f, int x1, int y1, int x2, int y2, pa_strptr sp, int id);

Creates a drop edit box in the window **f**, within rectangle **x1**, **y1**, **x2**, **y2**, for string list **sp**, with logical identifier **id**. The bounding rectangle for a drop edit box specifies its open mode, where the user has selected and dropped down the list of items. If the size specified is greater than or equal to the open size, as determined by **pa_dropboxsiz()**, then the entire dropbox will be presented. If the size is between the closed size and the open size, the system will attempt to work around the fact that the entire list cannot be dropped down. This is typically done by allowing the user to scroll though the list. If the size is less than the closed size, the dropbox will be clipped.

When a drop box string is selected, or enter is hit while editing, it sends the event **pa_etdrebox**. There is no other information associated with this event. Since the text is editable, it could be anything, and may not match one of the list entries. Instead, the program should use **pa_getwidgettext()** to retrieve the result of the edit.

Drop edit boxes default to a blank edit string. The idea of the drop edit box is that the user can simply use it as an edit box to enter the needed data, or drop down a list of preselected items. If you wish to make one of the string list items the default, or even a text that is not on the list, use the **pa_putwidgettext()** to initialize the edit field.

void pa_slidehorizsiz[g](FILE* f, int* w, int* h);

Finds the size of a horizontal scrollbar for window **f**. The required width is returned in **w**, and the required height in **h**. The height of a slider is chosen so that they match other slidebars used in the system. The width is a suggestion, and can be ignored.

void pa_slidehoriz[g](FILE* f, int x1, int y1, int x2, int y2, int mark, int id);

Creates a horizontal slider in window **f**, in bounding rectangle **x1**, **y1**, **x2**, **y2**, with **mark** number of tick marks, and a logical identifier **id**. Sliders give a convenient way to select from a range of values. The system will either size the slider to fit the given rectangle, or choose the slider representation that is as large as possible, but still fits the given rectangle. It is not guaranteed that the slider will fill the entire rectangle. This means that it is important for the entire background under the rectangle to be set to the standard background color.

When the slider is moved by the user, it generates a **pa_etsldpos** event. This event carries the new position of the slider, which is a ratioed **INT_MAX** number, from 0 to **INT_MAX**. 0 means the slider is at the extreme left, and **INT_MAX** means the slider is at the extreme right.

There is no guarantee as to when a slider generates its events. It can generate them as the slider is moved, or it may wait until the user moves, and then releases the slider to generate events. Sliders automatically update the position of the slider, and do not need to be set by the program.

The number of tick marks given by **mark** are evenly distributed across the slider. If **mark** is zero, then no tick marks are placed at all. Tick marks have no other effect besides appearing on the slider.

void pa_slidevertsiz[g](FILE* f, int* w, int* h);

Finds the size of a vertical slider for window **f**. The required width is returned in **w**, and the required height in **h**. The width of a slider is chosen so that they match other slidebars used in the system. The height is a suggestion, and can be ignored.

void pa_slidevert[g](FILE* f, int x1, int y1, int x2, int y2, int mark, int id);

Creates a vertical slider in window **f**, in bounding rectangle **x1**, **y1**, **x2**, **y2**, with mark number of tick marks, and a logical identifier **id**. Sliders give a convenient way to select from a range of values. The system will either size the slider to fit the given rectangle, or choose the slider representation that is as large as possible, but still fits the given rectangle. It is not guaranteed that the slider will fill the entire rectangle. This means that it is important for the entire background under the rectangle to be set to the standard background color.

When the slider is moved by the user, it generates a **etsldpos** event. This event carries the new position of the slider, which is a ratioed **INT_MAX** number, from 0 to **INT_MAX**. 0 means the slider is at the extreme top, and **INT_MAX** means the slider is at the extreme bottom.

There is no guarantee as to when a slider generates its events. It can generate them as the slider is moved, or it may wait until the user moves, and then releases the slider to generate events. Sliders automatically update the position of the slider, and do not need to be set by the program.

The number of tick marks given by **mark** are evenly distributed across the slider. If **mark** is zero, then no tick marks are placed at all. Tick marks have no other effect besides appearing on the slider.

void pa_tabbarsiz[g](FILE* f, pa_tabori tor, int cw, int ch, int* w, int* h, int* ox, int* oy);

Finds the size of a tabbar, in window **f**, with tab orientation **tor**, client width **cw**, and client height **ch**. The required width is returned in **w**, the height in **h**, and the client offset in **ox** and **oy**. The size of a tabbar is enough to hold the height of the labeling font (in whatever orientation), plus border areas, any selection scrolling arrows, and the client area.

void pa_tabbarclient[g](FILE* f, pa_tabori tor, int w, int h, int* cw, int* ch,int* ox, int* oy);

Finds the size of a tabbar client area, in window **f**, with tab orientation **tor**, tabbar width **w**, and tab bar height **h**. The client width is returned in **cw**, the height in **ch**, and the client offset in **ox** and **oy**. This procedure is used to find the client area for a specific size of tabbar.

void pa_tabbar[g](FILE* f, int x1, int y1, int x2, int y2, pa_strptr sp, pa_tabori tor, int id);

Creates a tab bar, in the window **f**, in the rectangle **x1**,**y1** to **x2**,**y2**, with string list **sp**, tab orientation tor, and logical identifier **id**. A tabbar gives the user a paradigm of a book with tabs on the side. The list of tabs, which are specified by the program, each generate events. The program establishes the view to be selected in the client area at the center of the tabbar, then it uses the tab events to switch the views in the client. If there is not enough area to display the full list of tabs, the system will allow the user to scroll through them with arrows.

To locate the tabbar and client, the **pa_tabbarsiz()** call is used to establish the size and client offset. Then, the client is offset into the tabbar. The client widgets or child windows must be created after the tabbar in order to be placed to the front of the stacking order.

When a tab is selected, it generates an **pa_ettabbar** event, which contains both the logical tabbar id and the number of the string list that was selected. This number will be the number in the string list. For example, the first string in the list will be 1, the second 2, etc.

A tab bar can have tabs on any of its sides. If a tab bar needs tabs on more than one side, the standard method is to overlay multiple tabs.

void pa_tabsel(FILE* f, int id, int tn);

Select tab in tab bar. Causes the tab **tn** in the tab bar **id** in the window **f**, to enter the selected state. **tn** is the number of the string list item to select. For example, the first string in the list will be 1, the second 2, etc.

void pa_alert(char* title, char* message);

Creates an alert dialog, with window title **title**, and client message **msg**. The alert dialog is a freestanding window that is placed at the front of the desktop stacking order. It is generally used to display an error, warning or other attention condition. The window title should tell the user what application is generating the alert, and the client message should give the error or warning. The user dismisses the dialog, and the program holds until the dialog completes.

void pa_querycolor(int* r, int* g, int* b);

Creates a color select dialog. The dialog "flows through" to set its parameters. When called, **r** contains the default red, **g** the default green, and **b** the default blue colors. These defaults are used to set the dialog default selection. When the user chooses a color, the same parameters return the new selection. If the user cancels, or leaves the default selection alone, the input colors will simply be left as the default output colors. The dialog is presented to the front of the desktop stacking order, and the program holds until the dialog is complete.

void pa_queryopen(char* s, int sl);

Creates an open file dialog. The dialog "flows through" to set its parameters. When called, **s** contains a default filename to open (which could be empty). This default is used to initialize the dialog default. When the user edits a filename, that is then returned in **s** as well, with maximum length **sl**. The input and output strings are not the same even if the user chooses the default. The result string must be disposed of by the caller, and the default string supplied to the dialog is allocated and deallocated entirely by the caller as well. The dialog is presented to the front of the desktop stacking order, and the program holds until the dialog is complete.

If the user cancels, an empty.

void pa_querysave(char* s, int sl);

Creates an save file dialog. The dialog "flows through" to set its parameters. When called, **s** contains a default filename to save (which could be empty). This default is used to initialize the dialog default. When the user edits a filename, that is then returned in **s** as well, with maximum length **sl**. The input and output strings are not the same even if the user chooses the default. The result string must be disposed of by the caller, and the default string supplied to the dialog is allocated and deallocated entirely by the caller as well. The dialog is presented to the front of the desktop stacking order, and the program holds until the dialog is complete.

If the user cancels, an empty string is returned.

void pa_queryfind(char* s, int sl, pa_qfnopts* opt);

Creates a find string dialog. The dialog "flows through" to set its parameters. When called, **s** contains a default search string (which could be empty). This default is used to initialize the dialog default. When the user edits a search, that is then returned in **s** as well, with a maximum length **sl**. The dialog is presented to the front of the desktop stacking order, and the program holds until the dialog is complete.

The find dialog may set one or more of several options from the set provided in **opt**. These are set before the call, and they are used to initialize the defaults in the dialog. When the dialog terminates, the new state of the options are returned in **opt** as well. If the dialog does not implement a particular option, then the input value will simply be copied to the output value without change.

If the user cancels, an empty string is returned.

void pa_queryfindrep(char* s, int sl, char* r, int rl, pa_qfropts* opt);

Creates a find/replace string dialog. The dialog "flows through" to set its parameters. When called, **s** contains a default search string (which could be null), and **r** contains the default replacement string (which could be null). This default is used to initialize the dialog default. When the user edits a search, that is then returned in **s** and **r** as well, with maximum lengths **sl** and **rl**. The input and output strings are not the same even if the user chooses the default. The result strings must be disposed of by the caller, and the default strings supplied to the dialog are allocated and deallocated entirely by the caller as well. The dialog is presented to the front of the desktop stacking order, and the program holds until the dialog is complete.

The find dialog may set one or more of several options from the set provided in **opt**. These are set before the call, and they are used to initialize the defaults in the dialog. When the dialog terminates, the new state of the options are returned in **opt** as well. If the dialog does not implement a particular option, then the input value will simply be copied to the output value without change.

If the user cancels, an empty string is returned for both **s** and **r**.

void pa_queryfont(FILE* f, int* fc, int* s, int* fr, int* fg, int* fb, int* br, int* bg, int* bb,
pa_qfteffects* effect);

Creates a query font dialog. The dialog "flows through" to set its parameters. When called, **fc** contains the font code, **s** contains the size, **fr**, **fg** and **fb** contain the foreground red, green and blue colors, **br**, **bg**, **bb** contains the background red, green and blue colors, and effect contains a set of font effects. These values are used to initialize the dialog defaults. The user then sets any or all of the parameters, and the results are copied back to the same output parameters. The dialog is presented to the front of the desktop stacking order, and the program holds until the dialog is complete.

If the dialog does not have a particular feature such as color set ability or one or more effects, the input for that parameter is simply copied to the output.

11.14 Events In widgets

See the description of the event record (11.12 “Events”) for the format of the entire record.

Event: pa_etbutton

The button with identifier **butid** was pressed.

Event: pa_etchkbox

The checkbox with identifier **chbxid** was selected.

Event: pa_etradbut

The radio button with identifier **radbid** was selected.

Event: pa_etsclull

The scrollbar with identifier **sclulid** had its up or left line button pressed.

Event: pa_etscldrl

The scrollbar with identifier **scldrid** had its down or right line button pressed

Event: pa_etsclulp

The scrollbar with identifier **sclupid** had its up or left page button pressed.

Event: pa_etscldrp

The scrollbar with identifier **scldpid** had its down or right page button pressed.

Event: pa_etsclpos

The scrollbar with identifier **sclid** was repositioned to **sclpos**. The value **sclpos** is INT_MAX ratio’ed, with 0 indicating top or left, and INT_MAX indicating bottom or right.

Event: pa_etedtbox

The editbox with identifier **edtbid** was given an enter key. This means that the text in the editbox is complete, and can be retrieved by **pa_etwidgettext()**.

Event: pa_etnumbox

The numselbox with the identifier **numbid** was entered with the number **numbsl**. **numbsl** directly corresponds to the number selected.

Event: pa_etlstbox

The listbox with the identifier **lstbid** was entered with the logical select **lstbsl**. The logical select **lstbsl** gives the number of the string selected in the order used to create the listbox, with 1 indicating the first string, 2 the second, etc.

Event: pa_etdrpbox

The listbox with the identifier **drpbid** was entered with the logical select **drpbsl**. The logical select **drpbsl** gives the number of the string selected in the order used to create the listbox, with 1 indicating the first string, 2 the second, etc.

Event: pa_etdrebox

The dropeditbox with identifier **drebid** was given an enter key. This means that the text in the editbox is complete, and can be retrived by **pa_getwidgettext()**.

Event: pa_etsldpos

The slider with identifier **sldpid** was repositioned to pos. The value **sldpos** is INT_MAX ratio'ed, with 0 indicating top or left, and INT_MAX indicating bottom or right.

Event: pa_ettabbar

The tabbar with the identifier **tabid** had a tab selected with the string number **tabsel**. tor indicates which in which string list the select occurred, top, bottom, left, right. **tabsel** indicates which string in that list was selected, with 1 being the first, 2 being the second, etc.

12 Sound Library

sound adds both a synthesizer interface via the MIDI standard, and the ability to play or input wave files. It implements a sequencer that programs the exact time at which each of the events occurs to make a complex combination of sounds.

Generation of sound for games and other uses can use the MIDI interface or the waveform interface. MIDI gives much more compact descriptions of complex sounds than waveforms, but in today's computers, that is not an issue. A game or other program may use waveforms for all of its sound outputs, even if MIDI was originally used to generate the waveform.

The MIDI interface is defined as a serial interface, but the target of a MIDI port can be a standard serial MIDI daisy chain, another type of interface that carries MIDI commands (such as USB), or simply terminate in an internal sound card or a software synthesizer.

Older keyboards with MIDI acted as both controllers and synthesizers.

Now with software based synthesizers, hardware that is a MIDI controller only is gaining popularity, including compact controllers with drumpads and rotary controllers.

12.1 Ports

Sound has both MIDI synthesizer ports and waveform device ports, and input and output ports of each. The number of synthesizer and wave ports are found by

pa_synthout()	The number of synthesizer output ports.
pa_synthin()	The number of synthesizer input ports.
pa_waveout()	The number of waveform output ports.
pa_wavein()	The number of waveform input ports.

The ports are arranged in order so that port 1 is the default input or output port. These are also found as:

```
#define PA_SYNTH_OUT 1 /* the default output synth for host */
#define PA_SYNTH_IN  1 /* The default input from external synth */
#define PA_WAVE_IN   1 /* the default wave input for host */
#define PA_WAVE_OUT  1 /* the default output wave for host */
```

Each port has a name, descriptive of its function or place in the computer. This is used to present a list of ports to the user, who selects which one is wanted. The name of each port is found as:

pa_synthoutname(p,n,l)	Return the name of a synthesizer output port.
pa_synthinname(p,n,l)	Return the name of a synthesizer input port.
pa_waveoutname(p,n,l)	Return the name of a waveform output port.
pa_waveinname(p,n,l)	Return the name of a waveform input port.

Where **p** is the port number, **n** is a string buffer, and **l** is the length of the buffer.

12.2 MIDI Output: Composition Interface

Sound has a composition interface for MIDI output devices. Typically, a computer has two output ports, the sound card internal to the computer with an onboard synthesizer, and the external MIDI jack. Alternately, the sound card synthesizer may be replaced by a software synthesizer. A synthesizer output

is opened with **pa_opensynthout(p)**, where **p** is the synthesizer port. It can be closed with **pa_closesynthout(p)**, where **p** is the synthesizer port. All synthesizer ports are automatically closed when the program closes.

12.3 Notes

```
typedef int pa_note; /* 1..128 note number for midi */
```

The basic work of making music is playing notes. MIDI can play 128 notes, numbered from 1 to 128. This ranges in frequency from 8 Hertz, or cycles per second, to 12 Kilohertz. This covers the range of human hearing associated with music. MIDI can also change each note in frequency enough to move it to the note next to it (and then some), so MIDI is able to reach any frequency desired.

Humans perceive a frequency that is 4 times higher as being only twice as high. If a musical note is doubled in frequency, it will be perceived as the same note one octave higher. There are twelve notes in an octave. In the lowest octave, they are:

Symbol	Number
note_c	1
note_c_sharp	2
note_d_flat	2
note_d	3
note_d_sharp	4
note_e_flat	4
note_e	5
note_f	6
note_f_sharp	7
note_g_flat	7
note_g	8
note_g_sharp	9
note_a_flat	9
note_a	10
note_a_sharp	11
note_b_flat	11
note_b	12

The bases of the octaves are:

Symbol	Number
octave_1	0
octave_2	12
octave_3	24
octave_4	36
octave_5	48
octave_6	60
octave_7	72
octave_8	84
octave_9	96
octave_10	108
octave_11	120

So any note in any octave can be found by:

note+octave

For example, C in the 6th Octave:

PA_NOTE_C+PA_OCTAVE_6

Notes are activated in MIDI by the **pa_noteon(p,t,c,n,v)** and notes are deactivated by **pa_noteoff(p,t,c,n,v)**. Each of these calls takes:

- A Port **p.**
- A time **t.**
- A channel **c.**
- A note **n.**
- A volume **v.**

Each of these parameters will be presented separately. The time to play will be discussed below. For now, it can be zero, which means "play it now". The channel is the instrument type to play it to, for instance, a piano, or an organ, or a tuba. The note is the logical note number we saw above, one of the 128 MIDI notes. The volume gives the volume the particular note is to be played at. A piano note can be louder if hit harder.

A note can either last forever, until turned off with **pa_noteoff()**, or it can stop on its own. For example, an organ plays as long as you hold the key down, but a string instrument plays a note when the string is plucked, then dies away. **pa_noteoff()** need not be used for these instruments, but can still be used to cause the note to be "clipped" off early, much as if the player put a hand on the string to stop it. Similarly, a **pa_noteon()** can be used to restart the note, even while it is playing.

12.4 Channels and Instruments

```
typedef int pa_channel;    /* 1..16   channel number */
typedef int pa_instrument; /* 1..128  instrument number */
```

MIDI has from 1 to 16 logical channels, indexed by a logical channel number. Although there are 128 instruments, only one can be played at any one time. To play an instrument, it must be assigned to a channel. This is done with **pa_instchange(p,t,c,i),** where i is the instrument.

The instruments that can be assigned to a given channel range from1 to 128, and **sndlib** provides symbols for each of them. The instruments are grouped, with 8 instruments per group, for a total of 16 groups.

Timing

S = Self timed
C = Configurable timing via noteon/noteoff

1.1.1 Piano Group

Instrument	Symbol	Number	Timing
Acoustic Grand	inst_acoustic_grand	1	S
Bright Acoustic	inst_bright_acoustic	2	S
Electric Grand	inst_electric_grand	3	S
Honky Tonk	inst_honky_tonk	4	S
Electric Piano 1	inst_electric_piano_1	5	S
Electric Piano 2	inst_electric_piano_2	6	S
Harpsichord	inst_harpsichord	7	S
Clavinet	inst_clavinet	8	S

1.1.2 Chromatic percussion Group

Instrument	Symbol	Number	Timing
celesta	inst_celesta	9	S
glockenspiel	inst_glockenspiel	10	S
music box	inst_music_box	11	S
vibraphone	inst_vibraphone	12	S
marimba	inst_marimba	13	S
xylophone	inst_xylophone	14	S
tubular bells	inst_tubular_bells	15	S
dulcimer	inst_dulcimer	16	S

1.1.3 Organ Group

Instrument	Symbol	Number	Timing
drawbar organ	inst_drawbar_organ	17	C
percussive organ	inst_percussive_organ	18	C
rock organ	inst_rock_organ	19	C
church organ	inst_church_organ	20	C
reed organ	inst_reed_organ	21	C
accoridan	inst_accoridan	22	C
harmonica	inst_harmonica	23	C
tango accordion	inst_tango_accordian	24	C

1.1.4 Guitar Group

Instrument	Symbol	Number	Timing
nylon string guitar	inst_nylon_string_guitar	25	S
steel string guitar	inst_steel_string_guitar	26	S
electric jazz guitar	inst_electric_jazz_guitar	27	S
electric clean guitar	inst_electric_clean_guitar	28	S
electric muted guitar	inst_electric_muted_guitar	29	S
overdriven guitar	inst_overdriven_guitar	30	S
distortion guitar	inst_distortion_guitar	31	S
guitar harmonics	inst_guitar_harmonics	32	S

1.1.5 Bass Group

Instrument	Symbol	Number	Timing
acoustic bass	inst_acoustic_bass	33	S
electric bass finger	inst_electric_bass_finger	34	S
electric bass pick	inst_electric_bass_pick	35	S
fretless bass	inst_fretless_bass	36	S
slap bass 1	inst_slap_bass_1	37	S
slap bass 2	inst_slap_bass_2	38	S
synth bass 1	inst_synth_bass_1	39	S
synth bass 2	inst_synth_bass_2	40	S

1.1.6 Solo strings Group

Instrument	Symbol	Number	Timing
violin	inst_violin	41	S
viola	inst_viola	42	S
cello	inst_cello	43	S
contrabass	inst_contrabass	44	S
tremolo strings	inst_tremolo_strings	45	S
pizzicato strings	inst_pizzicato_strings	46	S
orchestral strings	inst_orchestral_strings	47	S
timpani	inst_timpani	48	S

1.1.7 Ensemble Group

Instrument	Symbol	Number	Timing
string ensemble 1	inst_string_ensemble_1	49	S
string ensemble 2	inst_string_ensemble_2	50	S
synthstrings 1	inst_synthstrings_1	51	S
synthstrings 2	inst_synthstrings_2	52	S
choir aahs	inst_choir_aahs	53	S
voice oohs	inst_voice_oohs	54	S
synth voice	inst_synth_voice	55	S
orchestra hit	inst_orchestra_hit	56	S

1.1.8 Brass Group

Instrument	Symbol	Number	Timing
trumpet	inst_trumpet	57	C
trombone	inst_trombone	58	C
tuba	inst_tuba	59	C
muted trumpet	inst_muted_trumpet	60	C
french horn	inst_french_horn	61	C
brass section	inst_brass_section	62	C
synthbrass 1	inst_synthbrass_1	63	C
synthbrass 2	inst_synthbrass_2	64	C

1.1.9 Reed Group

Instrument	Symbol	Number	Timing
soprano sax	inst_soprano_sax	65	C
alto sax	inst_alto_sax	66	C
tenor sax	inst_tenor_sax	67	C
baritone sax	inst_baritone_sax	68	C
oboe	inst_oboe	69	C
english horn	inst_english_horn	70	C
bassoon	inst_bassoon	71	C
clarinet	inst_clarinet	72	C

1.1.10 Pipe Group

Instrument	Symbol	Number	Timing
piccolo	inst_piccolo	73	C
flute	inst_flute	74	C
recorder	inst_recorder	75	C
pan flute	inst_pan_flute	76	C
blown bottle	inst_blown_bottle	77	C
skakuhachi	inst_skakuhachi	78	C
whistle	inst_whistle	79	C
ocarina	inst_ocarina	80	C

1.1.11 Synth lead Group

Instrument	Symbol	Number	Timing
lead 1 square	inst_lead_1_square	81	C
lead 2 sawtooth	inst_lead_2_sawtooth	82	C
lead 3 calliope	inst_lead_3_calliope	83	C
lead 4 chiff	inst_lead_4_chiff	84	C
lead 5 charang	inst_lead_5_charang	85	C
lead 6 voice	inst_lead_6_voice	86	C
lead 7 fifths	inst_lead_7_fifths	87	C
lead 8 bass lead	inst_lead_8_bass_lead	88	C

1.1.12 Synth pad Group

Instrument	Symbol	Number	Timing
pad 1 new age	inst_pad_1_new_age	89	C
pad 2 warm	inst_pad_2_warm	90	C
pad 3 polysynth	inst_pad_3_polysynth	91	C
pad 4 choir	inst_pad_4_choir	92	C
pad 5 bowed	inst_pad_5_bowed	93	C
pad 6 metallic	inst_pad_6_metallic	94	C
pad 7 halo	inst_pad_7_halo	95	C
pad 8 sweep	inst_pad_8_sweep	96	C

1.1.13 Synth effects Group

Instrument	Symbol	Number	Timing
fx 1 rain	inst_fx_1_rain	97	C
fx 2 soundtrack	inst_fx_2_soundtrack	98	C
fx 3 crystal	inst_fx_3_crystal	99	C
fx 4 atmosphere	inst_fx_4_atmosphere	100	C
fx 5 brightness	inst_fx_5_brightness	101	C
fx 6 goblins	inst_fx_6_goblins	102	C
fx 7 echoes	inst_fx_7_echoes	103	C
fx 8 sci fi	inst_fx_8_sci_fi	104	C

1.1.14 Ethnic Group

Instrument	Symbol	Number	Timing
sitar	inst_sitar	105	S
banjo	inst_banjo	106	S
shamisen	inst_shamisen	107	S
koto	inst_koto	108	S
kalimba	inst_kalimba	109	S
bagpipe	inst_bagpipe	110	S
fiddle	inst_fiddle	111	S
shanai	inst_shanai	112	S

1.1.15 Percussive Group

Instrument	Symbol	Number	Timing
tinkle bell	inst_tinkle_bell	113	S
agogo	inst_agogo	114	S
steel drums	inst_steel_drums	115	S
woodblock	inst_woodblock	116	S
taiko drum	inst_taiko_drum	117	S
melodic tom	inst_melodic_tom	118	S
synth drum	inst_synth_drum	119	S
reverse cymbal	inst_reverse_cymbal	120	S

1.1.16 Sound effects Group

Instrument	Symbol	Number	Timing
guitar fret noise	inst_guitar_fret_noise	121	C
breath noise	inst_breath_noise	122	C
seashore	inst_seashore	123	C
bird tweet	inst_bird_tweet	124	C
telephone ring	inst_telephone_ring	125	C
helicopter	inst_helicopter	126	C
applause	inst_applause	127	C
gunshot	inst_gunshot	128	C

1.1.1 Drum channel

When MIDI starts up, all channels are assigned logical instrument number 1, an acoustical grand piano, with the exception of channel 10.

The MIDI channel system allows an “arrangement” to be created from different instruments. Each channel is configured with an instrument, then used to play a sequence. The computer can quickly change instruments during the music, and start an entirely different kind of music without skipping a beat. It is good practice not to count on an instrument being able to complete a note that is playing if it is swapped out of its channel for another instrument.

Channel 10 is an exception. This channel is always reserved for percussion (or drum) sounds. In this channel, the notes sent have a special meaning. Each note selects a different instrument. The instruments in the drum channel are always self timed.

Instrument	Symbol	Note number
Acoustic bass drum	note_acoustic_bass_drum	35
Bass drum 1	note_bass_drum_1	36
Side stick	note_side_stick	37
Acoustic snare	note_acoustic_snare	38
Hand clap	note_hand_clap	39
Electric snare	note_electric_snare	40
Low floor tom	note_low_floor_tom	41
Closed hi hat	note_closed_hi_hat	42
High floor tom	note_high_floor_tom	43
Pedal hi hat	note_pedal_hi_hat	44
Low tom	note_low_tom	45
Open hi hat	note_open_hi_hat	46
Low mid tom	note_low_mid_tom	47
Hi mid tom	note_hi_mid_tom	48
Crash cymbal 1	note_crash_cymbal_1	49
High tom	note_high_tom	50
Ride cymbal 1	note_ride_cymbal_1	51
Chinese cymbal	note_chinese_cymbal	52
Ride bell	note_ride_bell	53
Tambourine	note_tambourine	54
Splash cymbal	note_splash_cymbal	55
Cowbell	note_cowbell	56
Crash cymbal 2	note_crash_cymbal_2	57
Vibraslap	note_vibraslap	58
Ride cymbal 2	note_ride_cymbal_2	59
Hi bongo	note_hi_bongo	60
Low bongo	note_low_bongo	61
Mute hi conga	note_mute_hi_conga	62
Open hi conga	note_open_hi_conga	63
Low conga	note_low_conga	64
High timbale	note_high_timbale	65
Low timbale	note_low_timbale	66
High agogo	note_high_agogo	67
Low agogo	note_low_agogo	68
Cabasa	note_cabasa	69
Maracas	note_maracas	70
Short whistle	note_short_whistle	71
Long whistle	note_long_whistle	72

Short guiro	note_short_guiro	73
Long guiro	note_long_guiro	74
Claves	note_claves	75
Hi wood block	note_hi_wood_block	76
Low wood block	note_low_wood_block	77
Mute cuica	note_mute_cuica	78
Open cuica	note_open_cuica	79
Mute triangle	note_mute_triangle	80
Open_triangle	note_open_triangle	81

The same instrument can be assigned to multiple channels. This allows an instrument harmonize with itself, playing overlapping notes.

12.5 Volume

The volume can be set for each individual note. Volume can be set for each channel by **pa_volsynthchan(p,t,c,v)**.

The volume is "**INT_MAX** ratioed". It exists as a value from 0 to **INT_MAX**, where 0 off (no volume) and **INT_MAX** is full on. It is not decibel compensated, meaning that **INT_MAX** div 2 is not half volume.

Balance between left and right can be set for each channel with **balance(p,t,c,b)**. It's still **INT_MAX** ratioed, except that 0 means middle, **INT_MAX** means full right, and -**INT_MAX** means full left.

12.6 Time and the Sequencer

sound MIDI does not have a concept of time built into the protocol. All notes or events sent to the MIDI port are assumed to happen "Now", unless an external sequencer is installed.

sound has sequencer support that is used by setting a time on each event call. If the time is 0, it means to send the note or event to the MIDI port immediately, otherwise the sequencer schedules the event to occur at the indicated time.

To start **sound**'s sequencer, the **pa_starttimeout()** is used, which starts a 100us counter running (it ticks every 10,000th of a second). Then, each time is specified relative to that running timer. The current time on the sequencer can be found with **pa_curtimeout()**, so the required time can be specified as an offset from that:

```
pa_curtime()+10000
```

means a time that is one second in the future.

This example dumps a "fanfare" into the MIDI port using the sequencer. It will be played using the time specified in the **pa_noteon()** and **pa_noteoff()** calls.

```
#include <sound.h>

#define OSEC 10000 /* one second */

int main(void)

{

   pa_starttimeout(); /* start sequencer */

   pa_noteon( PA_SYNTH_OUT,  0,                   1,
              PA_NOTE_C+PA_OCTAVE_6, INT_MAX);
   pa_noteoff(PA_SYNTH_OUT, PA_CURTIME+OSEC*2,  1,
              PA_NOTE_C+PA_OCTAVE_6, INT_MAX);
   pa_noteon( PA_SYNTH_OUT,  PA_CURTIME+OSEC*3,  1,
              PA_NOTE_D+PA_OCTAVE_6, INT_MAX);
   pa_noteoff(PA_SYNTH_OUT, PA_CURTIME+OSEC*4,  1,
              PA_NOTE_D+PA_OCTAVE_6, INT_MAX);
   pa_noteon( PA_SYNTH_OUT,  PA_CURTIME+OSEC*5,  1,
              PA_NOTE_E+PA_OCTAVE_6, INT_MAX);
   pa_noteoff(PA_SYNTH_OUT, PA_CURTIME+OSEC*6,  1,
              PA_NOTE_E+PA_OCTAVE_6, INT_MAX);
   pa_noteon( PA_SYNTH_OUT,  PA_CURTIME+OSEC*7,  1,
              PA_NOTE_F+PA_OCTAVE_6, INT_MAX);
   pa_noteoff(PA_SYNTH_OUT, PA_CURTIME+OSEC*8,  1,
              PA_NOTE_F+PA_OCTAVE_6, INT_MAX);
   pa_noteon( PA_SYNTH_OUT,  PA_CURTIME+OSEC*9,  1,
              PA_NOTE_E+PA_OCTAVE_6, INT_MAX);
   pa_noteoff(PA_SYNTH_OUT, PA_CURTIME+OSEC*10, 1,
              PA_NOTE_E+PA_OCTAVE_6, INT_MAX);
   pa_noteon( PA_SYNTH_OUT,  PA_CURTIME+OSEC*11, 1,
              PA_NOTE_D+PA_OCTAVE_6, INT_MAX);
   pa_noteoff(PA_SYNTH_OUT, PA_CURTIME+OSEC*13, 1,
              PA_NOTE_D+PA_OCTAVE_6, INT_MAX);

}
```

The fanfare plays, and the program goes on to other work, or waits for the sequenced time to pass by setting a timer to the time required to finish it, or uses **pa_waitwave().**

If a note is output (or other action) with a 0 time while the sequencer is running, it will still occur immediately. Time 0 always means "now". The sequencer is a "flow through" model. Actions and notes can be timed with the sequencer, or by the program, or any combination thereof.

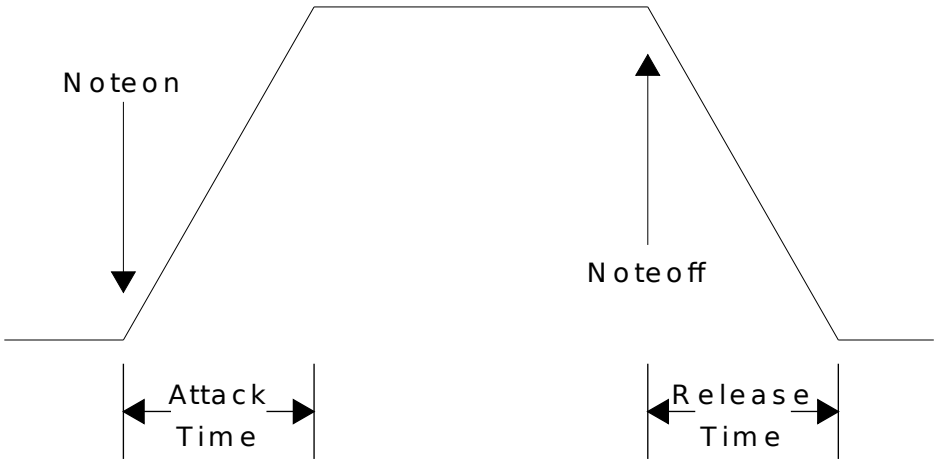
When the sequencer is no longer required, **pa_stoptimeout()** stops it. Doing that can save processor time, and free up system timers.

12.7 Effects

There are many effects in MIDI that can be applied to output notes. However, there is no requirement for the system to implement them. Few of the effects are implemented on most computer sound cards or software synthesizers.

Function	Description
pa_attack(p,t,c,at)	Adjusts the "attack time" **at** of each note.
pa_release(p,t,c,rt)	Adjusts the release or "decay" time **rt** of the note.
pa_reverb(p,t,c,r)	Sets the amount of reverberation **r**, or a series of repetitions of the note with delay.
pa_vibrato(p,t,c,v)	Sets the vibrato **v**, which is a pulsating pitch change.
pa_chorus(p,t,c,r)	Sets the chorus effect **r,** which is an echo of the same note with a delay and possible pitch change..
pa_phaser(p,t,c,ph)	Sets the phaser effect **ph**, which is a series of peaks and valleys in the frequency spectrum of the note.
pa_brightness(p,t,c,b)	Sets the brightness **b**, or VCF cutoff frequency.
pa_timbre(p,t,c,tb)	Sets the “tone color” **tb**.
pa_aftertouch(p,t,c,n,at)	Sets the amount of time or pressure used to sustain a key pressed. **at** is the time or pressure value.
pa_pressure(p,t,c,pr)	Sets the amount of pressure **p** applied to a key.
pa_legato(p,t,c,b)	Sets the note to be played shorter than normal. **b** is the value.
pa_portamento(p,t,c,b)	Sets the note to “slide” or smoothly change into the next note. **b** is the value.

Where **p** is the port, **t** is the time, and **c** is the channel. Note that all linear values are INT_MAX ratioed.



Attack and release times control the basic envelope of notes.

Some missing effects can be simulated by other means. As an example, **pa_release** control can be emulated by putting the instrument to control in its own channel, sounding the note, then lowering the volume in steps until 0, then turning the note off.

12.8 Pitch Changes

If a frequency is needed that is not exactly on a note, it can be “bent” with **pa_pitch(p,t,c,pt),** where **p** is the port, **t** is the time, **c** is the channel, and **pt** is the pitch offset. The pitch change is none for 0, and by default, one note up or down. In other words, the default pitch change range is one note up or down. A D note can be bent downwards to C, or upwards to E. The term "bend" comes from bending a string to change the note.

The default range of pitch changes can also be changed, by **pa_pitchrange(p,t,c,v),** where p is the port, t is time, c is the channel, and v is the pitch range value. The range is a ratioed 0..**INT_MAX**. 0 means no pitch range at all (disabled), and **INT_MAX** means the full 128 notes worth of pitch range. What you pick up with total range, you lose in fine control. If the pitch range is **INT_MAX**, each step of pitch change is going to be very coarse.

12.9 MIDI Input/Output: Structure Interface

The other way to format MIDI I/O is by keeping the MIDI messages in structures or records. This is the method you use when you are just dealing with MIDI messages as data to be transferred from place to place. When reading from a MIDI interface, you have to use this format, since you don’t know what the incoming message is. With the composition interface, you know beforehand what message you are going to construct. The MIDI structure is:

```
typedef int pa_note;       /* 1..128  note number for midi */
typedef int pa_channel;    /* 1..16   channel number */
typedef int pa_instrument; /* 1..128  instrument number */

/* sequencer message types. each routine with a sequenced option has
   a sequencer message assocated with it */
typedef enum {
    st_noteon, st_noteoff, st_instchange, st_attack, st_release,
    st_legato, st_portamento, st_vibrato, st_volsynthchan,
    st_porttime, st_balance, st_pan, st_timbre, st_brightness,
    st_reverb, st_tremulo, st_chorus, st_celeste, st_phaser,
    st_aftertouch, st_pressure,s t_pitch, st_pitchrange, st_mono,
    st_poly, st_playsynth, st_playwave, st_volwave
} pa_seqtyp;

/* sequencer message */
typedef struct pa_seqmsg {

    struct pa_seqmsg* next; /* next message in list */
    int               port; /* port to which message applies */
    int               time; /* time to execute message */
    pa_seqtyp         st;   /* type of message */
    union {

        /* st_noteon st_noteoff st_aftertouch st_pressure */
        struct { pa_channel ntc; pa_note ntn; int ntv; };
        /* st_instchange */
        struct { pa_channel icc; pa_instrument ici; };
        /* st_attack, st_release, st_vibrato, st_volsynthchan,
           st_porttime, st_balance, st_pan, st_timbre, st_brightness,
           st_reverb, st_tremulo, st_chorus, st_celeste, st_phaser,
           st_pitch, st_pitchrange, st_mono */
        struct { pa_channel vsc; int vsv; };
        /* st_poly */ pa_channel pc;
        /* st_legato, st_portamento */
        struct { pa_channel bsc; int bsb; };
        /* st_playsynth */ int sid;
        /* st_playwave */ int wt;
        /* st_volwave */ int wv;

    };

} pa_seqmsg;

/* pointer to message */
typedef pa_seqmsg* pa_seqptr;
```

To write a MIDI message structure to an output port, use **pa_wrsynth(p, sp)**, where **p** is the port, and **sp** is a pointer to a MIDI message structure. To read from a MIDI input port to a MIDI message structure, use **pa_rdsynth(p, sp)**, where **p** is the port, and **sp** is a pointer to a MIDI message structure.

Although MIDI outputs will tolerate any speed of messages on output, input MIDI devices will generate messages at whatever the rate the external device requires. Its possible that the input device will wait for your program to read messages, say if the input is supplied from the file, otherwise your code must be prepared to deal with real time devices that need to be read at any possible rate.

MIDI devices are considered to have a maximum message rate of about 1000 messages per second. Thus your program should ideally be able to process messages in no less than a millisecond. This is certainly within the capability of modern computers, which can execute from 1,000 to as much as 1,000,000 assembly instructions per millisecond. When reading from MIDI inputs, simply be aware that failure to read in a timely manner will cause data loss.

12.10 Prerecorded MIDI Files

We don't have to make all our MIDI commands on the fly. In fact, we can forget doing any MIDI, and just play back prerecorded MIDI files. To reduce latency with playing MIDI files, the file must be preloaded with **pa_loadsynth(s, sf)**, where **s** is the logical synthesizer cache number, and **sf** is a string with the name of the MIDI file. To load a MIDI file can mean anything from just storing the name of the file, to reading and converting the MIDI messages the file contains into memory. There are at least 10 positions in the MIDI synthesizer cache, numbered 1 to n. The cached file is played with **pa_playsynth(p,t,s).** where **p** is the output port, **t** is the time to play it, and **s** is the logical cache number. The format is system defined. Note that even though the prerecorded MIDI file has its own timing, it is played relative to the clock start position that is indicated for it. To remove a MIDI file from the cache, use **pa_delsynth(s)**, where **s** is the logical cache number of the MIDI file.

Because it is difficult or impossible to determine when a played MIDI file will end, the function **pa_waitsynth(p)** can be used, where **p** is the synthesizer output port. It blocks until the given synthesizer completes, that is, runs out of messages to process. Thus after scheduling a MIDI file to play, waiting for the synthesizer playing it to finish will play the entire file.

12.11 Waveform Input/Output

Waveform files are how we get arbitrary sounds into and out of the computer. Anything, for any length, can be played via the waveform files. Waveform outputs go out via their own ports separate from MIDI ports. Like MIDI, however, they are selected via logical numbers from 1 to n, where n is the maximum number of waveform output devices on the computer.

Waveforms can be both input and output with sound. To directly input or output waveforms, the I/O must be performed in real time, that is, the program must keep up with the speed of external devices.

The basic calls for waveform devices are:

Function	Description
pa_wavein()	Find number of waveform input devices.
pa_waveout(p)	Find number of waveform output devices.
pa_openwavein(p)	Open Waveform input device.
pa_closewavein(p)	Close waveform input device.
pa_openwaveout(p)	Open Waveform input device.
pa_closewaveout(p)	Close waveform input device.

Where **p** is the port number for the waveform device.

Waveform devices have several important characteristics:

- Number of channels (mono, stereo, quad, etc).
- Sample rate.
- Number of bits used for each sample.
- Signed or unsigned format of each sample.
- Endian (byte order) of each sample.
- Floating point vs. integer format of samples.

You can count on a single sound card or device having a set format or a number of set formats, and having the input formats match the output formats. However, different hardware inputs and outputs on different devices or cards may have different formats. Its also possible for the computer to do “on the fly” conversion of sample formats for you. But if both the inputs and the outputs are capable of such conversions, how to you pick the appropriate sample format? You may specify conversion to/from 8 bits on both input and output, when both input and output are capable of 16 bits, thus not only cutting down the sound quality, but creating more processing work for no reason.

Because of this, sound uses the rules that:

- You examine the format of samples the input device can provide.
- You set the format of samples going to the output device.

Thus sound usually picks the best format for input, usually the one that involves no conversion, and has the highest quality sound samples, then your program will set what it wants for the output. If your program is copying from input to output, it should read the input sample format values, then set those same values to the output device.

Function to read input values	Function to set output values	Value
pa_chanwavein(p)	**pa_chanwaveout(p, c)**	Number of channels.
pa_ratewavein(p)	**pa_ratewaveout(p, r)**	Sample rate per second.
pa_lenwavein(p)	**pa_lenwaveout(p, l)**	Length of sample in bits.
pa_sgnwavein(p)	**pa_sgnwaveout(p, s)**	Signed or unsigned (**s**=1, **s**=0).
pa_endwavein(p)	**pa_endwaveout(p, e)**	Endian (**e**=1=big endian, **e**=0=little endian).
pa_fltwavein(p)	**pa_fltwaveout(p, f)**	Floating point (**f**=1=floating point, **f**=0=integer).

The output port must be open to set parameters. For input ports, the parameters can be read at any time.

Some of the parameters have no effect on sound quality. Endian mode and signed mode have no effect on quality. Again the idea is to match the formats to evade the need for conversion.

If you are only outputting samples, you can choose your format. The best format matches the characteristics of the machine you are working on. A little endian format for a little endian machine, etc. The best sound cards today will convert samples in hardware, on the fly.

For most uses, the gold standard is CD disc audio, which is 44,100 samples per second, 16 bits, unsigned integer samples, and stereo.

Waveform data is output by **pa_wrwave(p, b, l)**, where **p** is the port, **b** is the sample buffer, and **l** is the length. The data in the buffer must be formatted according to the parameters set by the previous parameter calls. The basic calculation for this is:

```
length=bit_length/8
if (bit_length%8) length++;
sample_size = length*channels;
```

The buffer size should be divisible by the sample size.

To keep up with the sample rate defined by the output channel, you have to supply data at least at the rate specified. For example, for the standard CD disc rate of 44.1 khz, with stereo 16 bit samples, you need:

16/8*2*44,100

or 176400 bytes per second. If you supply 512 byte sample buffers, thats 344 buffers per second, which is a rate today's computers can easily handle. To not experience interruptions or silent periods in the output, there must be no delay in outputting these buffers.

When **pa_wrwave()** is called, the data is automatically buffered for you. This must be, since if the sound is being played from the sample buffer, and **pa_wrwave()** does not return until all of the samples are played, you would have only a single sample time to prepare another full sample buffer.

If you are outputting prerecorded material, the buffer could be any length. For applications such as computer telephony, the latency, or amount of time that passes between the time a user says a word at the source and the time the receiving user hears it, must be tightly controlled. In telephony, the latency must be kept down below about 50ms. For the above 512 bye buffer example, the latency is:

512/(16/8)*2=128
1/44100=0.000022676 second
0.000022676*128=0.002902494 second latency.

So well below the 0.050 second requirement. To reduce the overhead in making repetitive calls, but still achieve the target latency, we can do:

0.050/ 0.000022676=2205 samples per buffer.

or:

2205*4=8819 bytes.

And still meet the latency requirements.

To read waveform data use **pa_rdwave(p,b,l)** where **p** is the input wave port, **b** is the sample buffer, and **l** is the length. It returns the length of data read, but this is only different from the buffer length if the channel is closed during the read, which does not apply for most devices.

The buffer sample format for input wave channels is the same as for output wave channels. Putting this all together, we can show a program that copies the standard input wave port to the standard output wave port:

```
#define BUFLEN 2048

unsigned char buff[2048]; * sample buffer *

/* open target ports */
pa_openwavein(PA_WAVE_IN);
pa_openwaveout(PA_WAVE_OUT);

/* transfer input parameters to output port */
pa_chanwaveout(PA_WAVE_OUT, pa_chanwavein(PA_WAVE_IN));
pa_ratewaveout(PA_WAVE_OUT, pa_ratewavein(PA_WAVE_IN));
pa_lenwaveout(PA_WAVE_OUT, pa_lenwavein(PA_WAVE_IN));
pa_sgnwaveout(PA_WAVE_OUT, pa_sgnwavein(PA_WAVE_IN));
pa_endwaveout(PA_WAVE_OUT, pa_endwavein(PA_WAVE_IN));
pa_fltwaveout(PA_WAVE_OUT, pa_fltwavein(PA_WAVE_IN));

/* transfer data continuously */
while (1) {

    pa_rdwave(sport, buff, BUFLEN);
    pa_wrwave(dport, buff, BUFLEN);

}
```

This is from the sample program connectwave.c.

You are not limited to just moving sound data to and from output or input sound ports. Computers have sufficient bandwidth to create, filter or perform complex operations on audio in real time. For example, this code generates a sine wave to the standard output wave device:

```
#define SIZEBUF 2048
#define PI 3.14159

double angle; /* start sine angle */

rate = 44100; /* set sample rate */

pa_openwaveout(PA_SYNTH_OUT);          /* open output wave port */
pa_chanwaveout(PA_SYNTH_OUT, 1);       /* one channel */
pa_ratewaveout(PA_SYNTH_OUT, 44100);   /* CD sample rate */
pa_lenwaveout(PA_SYNTH_OUT,  16);      /* 16 bits */
pa_sgnwaveout(PA_SYNTH_OUT,  TRUE);    /* signed */
pa_endwaveout(PA_SYNTH_OUT,  FALSE);   /* little endian */
pa_fltwaveout(PA_SYNTH_OUT,  FALSE);   /* integer */

while (1) {

    for (i = 0 ; i < SIZEBUF ; i++) {

        buf[i] = (short)(SHRT_MAX*sin(angle));
        angle += 2*PI*freq/rate;
        if (angle > 2 * PI) angle -= 2*PI;

    }
    pa_wrwave(PA_SYNTH_OUT, (byte*)buf, SIZEBUF);

}
```

This is from the sample program genwave.c.

12.12 Prerecorded Waveform Files

An entire waveform file can be played into an output waveform device. Waveform files are usually very system dependent, so the exact format of the file will be different for different systems.

Waveform files are cached to reduce latency in start time. Wave cache numbers are from 1 to n, where n is the maximum number of cached waveform files. At least 10 such files can loaded into cache at once.

A waveform file is loaded with **pa_loadwave(w,s)** where **w** is the logical cache number, and **s** is the string containing the wave file name. The waveform file is played with **pa_playwave(p,t,w),** where **p** is the waveform output port, **t** is the time to play it, and **w** is the cache number. A waveform file can be deleted from cache with **pa_delwave(w)**, where **w** is the logical cache number.

As with MIDI files, waveforms have their own timing, and are simply played relative to the indicated start time.

The playback volume for waveform files is adjusted with **pa_volwave(p,t,v),** where **p** is the output wave port, **t** is the time to change the volume, and **v** is the **INT_MAX** ratioed volume value.

It is possible that the implementation will only be able to play one waveform file at a time. In this case, the behavior will be to stop any currently playing waveform file and start the new one, if a new waveform play is ordered before the previous one has finished. High quality implementation will be capable of playing multiple waveform files at once, typically via mixing of the output.

When a waveform file is played, it is difficult or impossible to determine when the waveform playback will be done. For this the **pa_waitwave(p)** function exists. It will wait until all wave output activity on the given wave output port **p** is complete, then return.

12.13 Functions and Procedures in sound

void pa_starttimeout(void);

Start time for the output MIDI sequencer. Starts the sequencer running. If the sequencer is already running, it will be restarted at 0.

void pa_stoptimeout(void);

Stop output MIDI sequencer. Halts the sequencer timer, and releases it.

int pa_curtimeout(void);

Get current output sequencer time. Returns the current sequencer time, in 100 Microsecond counts. The count is guaranteed not to wrap for 24 hours.

void pa_starttimein(void);

Start time for the input MIDI sequencer. Starts the sequencer running/ If the sequencer is already running, it will be restarted at 0.

void pa_stoptimein(void);

Stop input MIDI sequencer. Halts the sequencer timer, and releases it.

int pa_curtimein(void);

Get current input sequencer time. Returns the current sequencer time, in 100 Microsecond counts. The count is guaranteed not to wrap for 24 hours.

int pa_synthout(void);

Find number of output synthesizers. Returns the total output synthesizers in the system.

int pa_synthin(void);

Find number of input synthesizers. Returns the total input synthesizers in the system.

void pa_opensynthout(int p);

Open output synthesizer. Opens the output synthesizer by the logical number **p**, where p is 1..**pa_synthout()**.

void pa_closesynthout(int p);

Close output synthesizer. Closes the output synthesizer by the logical number **p**.

void pa_opensynthin(void);

Open input synthesizer. Opens the input synthesizer by the logical number **p**, where p is 1..**pa_synthin()**.

void pa_closesynthin(void);

Close input synthesizer. Closes the input synthesizer by the logical number **p**.

void pa_noteon(int p, int t, pa_channel c, pa_note n, int v);

Start note. Starts a note for synthesizer **p**, in channel **c**, with note n, and 0..**INT_MAX** ratioed volume **v**.

void pa_noteoff(int p, int t, pa_channel c, pa_note n, int v);

Stop note. Stops a note for synthesizer **p**, in channel **c**, with note, and 0..**INT_MAX** ratioed volume **v**. **v** is usually ignored on a **noteoff**.

void pa_instchange(int p, int t, pa_channel c, pa_instrument i);

Change instrument. Changes the instrument assigned to a channel, for output port **p**, at time **t**, for channel **c**, to instrument **i**.

void pa_attack(int p, int t, pa_channel c, int at);

Set attack time. Sets the attack time for synthesizer output port p, at time **t**, for channel **c**, to 0..**INT_MAX** ratioed time **at**.

void pa_release(int p, int t, pa_channel c, int rt);

Set release time. Sets the release time for synthesizer output port **p**, at time **t**, for channel **c**, to 0..**INT_MAX** ratioed time **at**.

void pa_legato(int p, int t, pa_channel c, int b);

Set legato. Sets legato mode on or off, for synthesizer output port **p**, at time **t**, for channel **c**, to on/off value **b**.

void pa_portamento(int p, int t, pa_channel c, int b);

Set portamento. Sets portamento mode on or off, for synthesizer output port **p**, at time **t**, for channel c, to on/off value **b**.

void pa_vibrato(int p, int t, pa_channel c, int v);

Set vibrato. Sets vibrato amount, for synthesizer output port p, at time **t**, for channel **c**, to 0..**INT_MAX** ratioed value **v**.

void pa_volsynthchan(int p, int t, pa_channel c, int v);

Set volume for channel. Sets volume for channel, for synthesizer output port **p**, at time **t**, for channel **c**, to 0..**INT_MAX** ratioed value **v**.

void pa_porttime(int p, int t, pa_channel c, int v);

Set portamento time. Sets portamento time, for synthesizer output port **p**, at time **t**, for channel **c**, to 0..**INT_MAX** ratioed value **v**.

void pa_balance(int p, int t, pa_channel c, int b);

Set channel balance. Sets the right left balance for synthesizer output port **p**, at time **t**, for channel **c**, to -**INT_MAX**..**INT_MAX** ratioed value **b**. -**INT_MAX** is full left, **INT_MAX** is full right, and 0 is centered.

void pa_pan(int p, int t, pa_channel c, int b);

Set channel pan. Sets the right left pan for synthesizer output port **p**, at time **t**, for channel **c**, to -**INT_MAX**..**INT_MAX** ratioed value. -**INT_MAX** is full left, **INT_MAX** is full right, and 0 is centered.

void pa_timbre(int p, int t, pa_channel c, int tb);

Set timbre. Sets timbre amount, for synthesizer output port **p**, at time **t**, for channel **c**, to 0..**INT_MAX** ratioed value **tb**.

void pa_brightness(int p, int t, pa_channel c, int b);

Set brightness. Sets brightness amount, for synthesizer output port **p**, at time **t**, for channel **c**, to 0..**INT_MAX** ratioed value **b**.

void pa_reverb(int p, int t, pa_channel c, int r);

Set reverb. Sets reverb amount, for synthesizer output port **p**, at time **t**, for channel **c**, to 0..**INT_MAX** ratioed value **r**.

void pa_tremulo(int p, int t, pa_channel c, int tr);

Set tremulo. Sets tremulo amount, for synthesizer output port **p**, at time **t**, for channel **c**, to 0..**INT_MAX** ratioed value **tr**.

void pa_chorus(int p, int t, pa_channel c, int cr);

Set chorus. Sets chorus amount, for synthesizer output port **p**, at time **t**, for channel **c**, to 0..**INT_MAX** ratioed value **cr**.

void pa_celeste(int p, int t, pa_channel c, int ce);

Set celeste. Sets celeste amount, for synthesizer output port p, at time **t**, for channel **c**, to 0..**INT_MAX** ratioed value **ce**.

void pa_phaser(int p, int t, pa_channel c, int ph);

Set phaser. Sets phaser amount, for synthesizer output port **p**, at time **t**, for channel **c**, to 0..**INT_MAX** ratioed value **ph**.

void pa_aftertouch(int p, int t, pa_channel c, pa_note n, int at);

Set aftertouch. Sets aftertouch amount, for synthesizer output port p, at time **t**, for channel **c**, to 0..**INT_MAX** ratioed value **at**.

void pa_pressure(int p, int t, pa_channel c, int pr);

Set pressure. Sets pressure amount, for synthesizer output port **p**, at time **t**, for channel **c**, to 0..**INT_MAX** ratioed value **pr**.

void pa_pitch(int p, int t, pa_channel c, int pt);

Set pitch bend. Sets the pitch "bend", or change amount, for synthesizer output port **p**, at time **t**, for channel **c**, to -**INT_MAX**..**INT_MAX** ratioed value **pt**. **pt** value is -**INT_MAX** for full down range, **INT_MAX** for full up range, and 0 for neutral (on note) pitch. The amount of pitch range is set by the pitchrange procedure, and defaults to one note down and one note up.

void pa_pitchrange(int p, int t, pa_channel c, int v);

Set pitch bend range. Sets the total amount of pitch change that can be reached by the pitch command, for synthesizer output port **p**, at time **t**, for channel **c**, to 0..**INT_MAX** ratioed value **v**. 0 disables the pitch command, and **INT_MAX** allows it to reach all 128 notes of MIDI. Note that increasing the range of the pitch command decreases its resolution.

void pa_mono(int p, int t, pa_channel c, int ch);

Set mono mode. Sets mono mode for synthesizer output port **p**, at time **t**, for channel **c**, for the number of channels **ch**. See MIDI specification for details.

void pa_poly(int p, int t, pa_channel c);

Set polyphonic mode. Sets polyphonic mode for synthesizer output port **p**, at time **t**, for channel **c**. Reverses the effect of a mono operation.

void pa_loadsynth(int s, string sf);

Load synthesizer file into cache. **s** is the logical number of the cache position, from 1 to n where n is the maximum number of cache positions, and **sf** is a string giving the full filename and path of the synthesizer file. The synthesizer file remains in the cache until it is specifically removed by **pa_delsynth()**.

Caching of synthesizer files allows the system to remove any possible latency from the time the playing of a synthesizer file is ordered to the time it actually starts playing. The function may completely load the file into memory, or it may just save the filename.

From 1 to 10 synthesizer cache positions are guaranteed, but the implementation may provide more.

The extension, type and format of the synthesizer file is system dependent. More that one format may be allowed.

The filename may be fully pathed.

void pa_playsynth(int p, int t, int s);

Play MIDI synthesizer file. Plays the MIDI instructions from the file by the cache position by logical number **s**, for output synthesizer **p**, at time **t**.

void pa_delsynth(int s);

Delete synthesizer file from cache. **s** is the logical synthesizer id. The indicated cache entry is cleared, and the logical id can be reused.

void pa_waitsynth(int p);

Waits until all pending activity on synthesizer port **p** has finished, then returns.

void pa_wrsynth(int p, pa_seqptr sp);

Write a synthesizer structure at pointer **sp** to synthesizer port **p**. The time for the structure will be examined, and if zero, it will be output immediately. Otherwise it is inserted into the synthesizer pending list.

If the function will store the structure for later output, a copy of it will be created. The structure can be reused immediately by the caller.

void pa_void pa_rdsynth(int p, pa_seqptr sp);

Read a synthesizer structure at pointer **sp** from synthesizer port **p**. The function does not allocate the structure, it only fills it out. The time provided in the structure is that of the input synthesizer clock, which may be zero if it has not been started. The timing provided on input is up to the system. It can be provided by the synthesizer input clock, if that has been started. Alternately it could be set at the MIDI input device (so called “device timing”), or it could read it in with the MIDI stream.

If the synthesizer input clock is not started, the input time is set to zero on all samples.

int pa_waveout(void);

Find number of waveform output devices. Returns the total number of waveform output devices in the system.

void pa_int pa_wavein(void);

Find number of waveform input devices. Returns the total number of waveform input devices in the system.

void pa_openwaveout(int p);

Open waveform device. Opens the logical waveform device **p**, where **p** is 1..**pa_waveout()**.

void pa_closewaveout(int p);

Close waveform device. Closes the logical waveform device **p**.

void pa_void pa_loadwave(int w, string fn);

Loads a waveform file with the name **fn** to the logical waveform cache **w**. The waveform cache logical numbers are from 1 to n, where n is the maximum number of waveform files that can be cached. This is guaranteed to be at least 10 files.

The purpose of caching waveform files is to allow the system to reduce or eliminate latency of the start time of the waveform output. **pa_loadwave()** may do anything from completely load the waveform file into the memory, to simply saving the name of the file for later load.

The filename for the waveform file may be fully pathed.

void pa_playwave(int p, int t, int w);

Play waveform file. Plays the waveform file by the logical cache number **w**, for output waveform device **p**, at time **t**. If the time **t** is left off, it defaults to 0.

void pa_int pa_wavein(void);

Gives the number of waveform input devices.

void pa_volwave(int p, int t, int v);

Set waveform volume. Sets the output waveform device volume for logical device **p**, at time **t**, to 0..**INT_MAX** ratioed value **v**.

void pa_waitwave(int p);

Waits until all waveform activity on waveform output port **p** is complete. This function is typically used to wait until the end of a **pa_playwave()**, which is difficult or impossible to know when the playback is complete.

void pa_chanwaveout(int p, int c);

Sets the number of channels **c** on an output waveform device **p**. The device must be open.

void pa_ratewaveout(int p, int r);

Sets the rate **r**, in samples per second, for the output waveform device **p**.

void pa_lenwaveout(int p, int l);

Sets the bit length of samples to **l** for the output waveform device **p**. The bit length of samples is rounded up to the next 8 bits for the sample size in bytes. The total sample size is found by round(l, 8)*channels, where round is a function to round up to the nearest 8 bits. The device must be open.

void pa_sgnwaveout(int p, int s);

Sets the signed format **s** for output waveform device **p**. **s** is 1 is the sample format is signed, and 0 if it is not. Note that it makes no difference to the bit length what the signed or unsigned format is. The device must be open.

void pa_fltwaveout(int p, int f);

Sets the floating point or integer format **f** of the waveform port **p**. **f** is 1 if the samples are in floating point format, and 0 if in integer format. Note that it makes no difference to the bit length what the floating point or integer format is. The device must be open.

Commonly, only binary power multiples have floating point formats, IE, 2, 4, 8 bytes, etc. The most common formats in use are the IEEE 754 formats for 2 and 4 bytes (16 bits and 32 bits).

void pa_endwaveout(int p, int e);

Sets the endian format **e** for the waveform port **p**. **e** is 1 for big endian, and 0 for little endian. Note that it makes no difference to the bit length what the endian format is. The device must be open.

The endian format should be set to match either the incoming stream or the natural endian mode of the current machine.

void pa_wrwave(int p, byte* b, int l);

Write a group of waveform samples from buffer **b**, with length **l**, to the output waveform port **p.** The buffer size should be a multiple of the sample size in bytes, which is round(l, 8)*c where l

is the bit length of samples, c is the number of channels, and round rounds up the length to the nearest 8 bits. It is undefined as to what the effect is if the buffer is not evenly divisible by the sample size. The device must be open.

If the output wave device is real time, the caller is responsible for writing buffers to the device at a rate sufficient to satisfy the output without gaps or silent periods in the output stream. At the same time, the caller is responsible for insuring that the latency of the stream is sufficiently low, typically less than 50 milliseconds for telephony applications.

After the function is called, the buffer is free to be changed by the caller.

pa_wrwave() will block or not block the caller to satisfy its own output stream requirements. Typically this will be to copy the incoming data to a buffer that is sized for latency concerns, and the caller will only be blocked if the internal buffer overflows. Thus the size of the buffer used to call **pa_wrwave()** may or may not have any relationship to the internal buffer size.

void pa_openwavein(int p);

Opens the wave input device **p**.

void pa_closewavein(int p);

Closes the wave input device **p**.

int pa_chanwavein(int p);

Returns the number of channels in samples from waveform input port **p**.

int pa_ratewavein(int p);

Returns the sample rate, in samples per second, from waveform input port **p**.

int pa_lenwavein(int p);

Returns the bit length in samples from waveform input port **p**.

int pa_sgnwavein(int p);

Returns the signed or unsigned format of samples from waveform input port **p**. Returns 1 is the samples are signed, and 0 if unsigned.

int pa_endwavein(int p);

Returns the big or little endian format of samples from waveform input port **p**. Returns 1 is the samples are big endian, and 0 if little endian.

int pa_fltwavein(int p);

Returns the floating point format of samples from waveform input port **p**. Returns 1 if the samples are floating point, and 0 if integer.

int pa_rdwave(int p, byte* buff, int len);

Read a group of waveform samples to buffer **b**, with length **l**, from the input waveform port **p.** The buffer size should be a multiple of the sample size in bytes, which is round(l, 8)*c where l is the bit length of samples, c is the number of channels, and round rounds up the length to the

nearest 8 bits. It is undefined as to what the effect is if the buffer is not evenly divisible by the sample size. The device must be open.

If the input wave device is real time, the caller is responsible for reading buffers from the device at a rate sufficient to satisfy the input without overruns in the input stream (and resulting loss of data). At the same time, the caller is responsible for insuring that the latency of the stream is sufficiently low, typically less than 50 milliseconds for telephony applications.

The caller allocates the buffer. If the function itself buffers data, it will copy the data to the caller provided buffer.

pa_rdwave() will block or not block the caller to satisfy its own input stream requirements. Typically this will be to copy the incoming data from a buffer that is sized for latency concerns, and the caller will only be blocked if there is no data ready from the input device. Thus the size of the buffer used to call **pa_rdwave()** may or may not have any relationship to the internal buffer size.

void pa_synthoutname(int p, string n, int l);

Returns the name of the synthesizer output port **p** in the string **n**, with buffer length **l**. The string is returned zero terminated. The buffer provided must have sufficient length to contain the string and it's zero termination.

The string returned will include the name of the device connected to the port, and may include descriptive text as well. It will fit on a single line.

void pa_synthinname(int p, string name, int l);

Returns the name of the synthesizer input port **p** in the string **n**, with buffer length **l**. The string is returned zero terminated. The buffer provided must have sufficient length to contain the string and it's zero termination.

The string returned will include the name of the device connected to the port, and may include descriptive text as well. It will fit on a single line.

void pa_waveoutname(int p, string name, int l);

Returns the name of the waveform output port **p** in the string **n**, with buffer length **l**. The string is returned zero terminated. The buffer provided must have sufficient length to contain the string and it's zero termination.

The string returned will include the name of the device connected to the port, and may include descriptive text as well. It will fit on a single line.

void pa_waveinname(int p, string name, int l);

Returns the name of the waveform input port **p** in the string **n**, with buffer length **l**. The string is returned zero terminated. The buffer provided must have sufficient length to contain the string and it's zero termination.

The string returned will include the name of the device connected to the port, and may include descriptive text as well. It will fit on a single line.

13 Networking Library

network gives ANSI C the ability to transfer data over a network such as the internet. It does this by connecting ANSI C files to network resources. Because of the use of standard file mechanisms, few added calls are needed.

network supports both continuous streams of data as well as messaging. It supports both plain text and encrypted communications via SSL/TLS. It supports both configuration as a client and a server.

13.1 IPv4 and IPv6 addresses

IPv4 and IPv6 addresses have formats unique from each other. IPv4 addresses are represented in 32 bit unsigned longs, and IPv6 addresses are represented in 128 bits as two unsigned long long (64 bits each). This format was chosen because 128 bit variables are not universally implemented in ANSI C. 64 bit is implemented widely even on 32 bit machines. It is the most universal format.

13.2 Standard stream channels

Streaming data is a series of bytes sent to and from a network connection. The data is secured in that any errors in the data are either fixed automatically or the program is stopped. The data sent or received is guaranteed both to be correct and in the same order as sent. When data is received, the call will wait until all data requested is received. Any amount of time may pass before that occurs, which is why network connections typically go with multiple threads.

To open a new network connection, **opennet(a,p,s)** is used, where **a** is the IP address, **p** is the port, and **s** indicates the connection is to be secured. The file handle used to send and receive data from the connection is returned. To close network connections, the standard ANSI C **close** is used. **opennet()** uses an address/port pair to indicate the network address of the server, and the port within the server. The address of a server, as determined from its name in characters, is found with **addrnet(n,a)** where **n** is the name of the server or its IP address in dotted form, and **a** is the resulting IPv4 address.

When a remote network port is opened, **network** treats the connection as a single synchronous channel going to and from the remote resource. All of the standard ANSI C I/O functions can be applied to such network connections, including **printf()**, **fprintf()**, **fscanf()**, **fgets()**, etc.

When working with IPv6 addresses, the calls **opennetv6(ah,al,p,s)** and **addrnetv6(n,a)** are used. This is necessary because IPv6 are not a superset of IPv4 addresses, but have a new and different format.

13.3 Secure channels

A secure channel with streaming data encodes all of the data passing over the interface so that, even though the client and the server can read it, third parties on the network cannot. To set a network connection as secure, the “secure” flag is simply set in **opennet(a,p,s)** or **opennetv6(ah,al,p,s)**, that’s all that is required. There is also a set of certificate functions if you need to verify or examine the contents of certificates.

13.4 Message based communications

An alternative to stream based network communication is message based. In this model, a fixed length of data is sent and received. It may or may not be guaranteed to be delivered. The order in which it is

received is also not garanteed. All that is guaranteed is that if a message contains data errors, it will not be delivered. The sender is responsible for handling interruptions in the data flow.

Message based communication can be a better match for certain communications types. For example, an audio call or broadcast would not be a good match for a stream, because an error in the data might stall the channel attempting to fix data corruption, and that would corrupt the timing of the data, causing it to fall behind the sender. Using message based instead allows the application to implement its own error handling procedure.

A message channel is opened with **pa_openmsg(a,p,s)**, where **a** is the address, **p** is the port, and **s** indicates the channel is to be secured. The function returns the logical number of the message file. If IPv6 is to be used, the call is **pa_openmsgv6(ah,al,p,s)**.

Message channels do not use ANSI C file functions. Reading a message is done with **pa_rdmsg(fn,m,l)** where **fn** is the logical message channel id, **m** is a pointer to the message buffer, and **l** is the maximum byte length of the buffer. It returns the actual number of bytes read. Writing a message is done with **pa_wrmsg(fn,m,l)** where **fn** is the logical message channel id, **m** is the message buffer, and l is the size of the data in the buffer.

Message channels are closed with **pa_clsmsg(fn)**, where **fn** is the logical message channel id.

The maximum possible size of messages for a given address is found by **pa_maxmsg(a)**, where **a** is the IP address. For IPv6, the maximum is found with **pa_maxmsgv6(ah,al)**. For standard IP, 1500 bytes is a common maximum, but so called “jumbo packets” can reach 64kb.

13.5 Reliable messaging

Messaging is widely used for interprocessor communication in computer clusters. The network carrying the messages may only exist in the same machine, or same internal network. The hardware may implement guaranteed messaging, so that the application does not have to deal with handling errors in the data, or reordering of messages. The program can determine if this is true with the call **pa_relymsg(a)**, where **a** is the IP address. It returns 1 if the message channel is reliable, and 0 if not. For IPv6 connections, **pa_relymsgv6(ah,al)** is used.

Another way of looking at reliable messaging is that if an error were to occur in data transfer, it would be fatal, and not just discarding a bad message.

network assumes that if a message will be sent and received on the current machine without being transmitted over a wire, that it will be reliable. Outside of that, it needs an indication that the hardware supports reliable messaging.

13.6 Serving Connections

So far we have talked about establishing connections to outside servers. We can also act as a server, waiting for inbound connections on a port. After such a connection is established, the server can send and receive data just as for outbound connections.

To wait on a stream connection, the function **pa_waitnet(p,s)** is used, where **p** is the port number, and **s** indicates the connection is to be secured. The file pointer used to talk to the connection is returned. The IP address of the connection is the IP address of the machine the program is being run on, and it can

be either IPv4, IPv6, or both. Any number of threads in the same application can open server connections on the same port, but other programs in the same system will be blocked from opening connections with that port.

For serving message based connections, the function **pa_waitmsg(p,s)** is used. It returns the logical channel number. The connection is then read and written with **pa_rdmsg()** and **pa_wrmsg()**.

13.7 Managing certificates

When a secure connection is created, the connection is secured by public key encryption. To make a channel, either stream or message, both the public keys and a "certificate" are exchanged. The key makes sure that only data from the given server or client is being received, and the certificate allows verification of who is on the other side of the communication. It is essentially a series of fields of data from the creator of the certificate, such as its IP address, the name of the organization, etc., that has been signed using the communications keys.

To check that the certificate is valid, it can be checked against a known database of so called "trusted" certificates. This would be the case for say, a company you know and trust with a privately created certificate. For others, say an arbitrary web site, the method used is a "chain of trust". A chain of trust is a series of certificates traceable back to a central "certificate authority". Each company that participates in a chain of trust purchases a certificate from a provider who can vouch for the fact that the company purchasing the certificate is who they say they are. Thus you would get both the certificate for the company that purchased it, and the certificate for the issuing authority. Issuing authorities can issue certificates to companies that are themselves authorized to issue certificates, and so on. These certificate "chains" can be any length. The benefit of the system is that only a few certificates ultimately need be held by the program accessing the servers, and a connection can be verified by following the chain of certificates.

How you manage certificates depends on your type of program. It is possible to create your own certificates, and you would keep a cache or local store of certificates for such "private" machines. Programs that use many machines on the public internet would store a list of so-called "Certificate Authorities" or CAs certificates, then you would traverse the chain of certificates and check those certificates exist in the local cache. **network** does not manage a certificate cache. This is typically done with a hashing scheme.

A certificate for streams can be obtained by **pa_certnet(f, w, s, l)**, where **f** is the connection file, **w** is which certificate in the chain to get, from 1 to n, where n is the last certificate in the chain. **s** is the buffer for the certificate, and **l** is the length of the buffer. The function returns the number of bytes read for the certificate. If there is no certificate by that number, 0 is returned. For message based channels, **pa_certmsg(fn,w,s,l)** is used.

Certificates for the previous calls are returned as a block of encoded data. To actually examine the certificate data, it is broken down into a tree structured series of data fields, of the format:

```
/* name - value pair list */
typedef struct pa_certfield {

    string                name;     /* name of field */
    string                data;     /* content of field */
    int                   critical; /* is a critical X509 field */
    struct pa_certfield*  fork;     /* sublist */
    struct pa_certfield*  next;     /* next entry in list */

} pa_certfield, *pa_certptr;
```

Thus each data item in the certificate contains a name and data for that field. It may also be a branch (a subcategory) which contains any number of sublists. It may also be marked as an essential X509 data field.

A certificate tree for a given certificate in a stream connection can be read by **pa_certlistnet(f,w,l)** where **f** is the connection file, **w** is the number of the certificate in the chain, 1 to n, and **l** is a pointer to the resulting certificate data tree. For message connections, use the function **pa_certlistmsg(fn,w,l)**, where **fn** is the file id, **w** indicates which certificate, and **l** is the data tree.

After a certificate data tree is used, it can be recycled by **pa_certlistfree(l)**, where **l** is a pointer to the certificate tree. This will recycle all of the data entries in the tree.

13.8 Functions and Procedures in network

FILE* pa_opennet(unsigned long addr, int port, int secure);

The server is indicated by a logical address number **addr**, whose exact meaning and format is dictated by the network itself. A logical port number **port** selects which resource within the server is being accessed. For the internet, this is a fixed constant that gives the exact service being asked of the far server.

The file used to read and write the connection is returned.

FILE* pa_opennetv6(unsigned long long addrh, unsigned long long addrl, int port, int secure);

The server is indicated by a logical IPv6 address number **addrh** and **addrl**, whose exact meaning and format is dictated by the network itself. A logical port number **port** selects which resource within the server is being accessed. For the internet, this is a fixed constant that gives the exact service being asked of the far server.

The file used to read and write the connection is returned.

pa_opennet() uses IPv4 addressing. **pa_opennetv6()** uses IPv6 addressing.

void pa_addrnet(const string name, unsigned long* addr);

addrnet takes the logical name of a server in string **name** and finds the address number **addr** for it. Such names are formatted according to the needs of the network.

Network connections are ended by the end of the program, or by using the standard file **close()** procedure on the handle of the connection.

void pa_addrnetv6(const string name, unsigned long long* addrh, unsigned long long* addrl);

addrnet takes the logical name of a server in string **name** and finds the IPv6 address number **addrh** and **addrl** for it. Such names are formatted according to the needs of the network.

Network connections are ended by the end of the program, or by using the standard file **close()** procedure on the handle of the connection.

int pa_maxmsg(unsigned long addr);

Returns the maximum size of a message for the IP address **addr**.

int pa_maxmsgv6(unsigned long long addrh, unsigned long long addrl);

Returns the maximum size of a message for the IPv6 address **addrh** and **addrl**.

int pa_relymsg(unsigned long addr);

Returns true if a connection to the given address **addr** is “reliable” or without data loss or reordering of messages. If a data loss or reorder error does happen, however unlikely, the program will halt with error.

int pa_relymsgv6(unsigned long long addrh, unsigned long long addrl);

Returns true if a connection to the given IPv6 address **addrh** and **addrl** is “reliable” or without data loss or reordering of messages. If a data loss or reorder error does happen, however unlikely, the program will halt with error.

int pa_openmsg(unsigned long addr, int port, int secure);

Opens a message channel with the address **addr**. If **secure** is true, then the messages will be encrypted.

int pa_openmsgv6(unsigned long long addrh, unsigned long long addrl, int port, int secure);

Opens a message channel with the IPv6 address **addrh and addrl**. If **secure** is true, then the messages will be encrypted.

void pa_wrmsg(int fn, void* msg, unsigned long len);

Write a message to the message channel by logical id **fn**, The message is in the buffer **msg**, and the length of the message is in **len**. The message must be less than or equal to **pa_maxmsg()** or **pa_maxmsgv6()** in size.

int pa_rdmsg(int fn, void* msg, unsigned long len);

Read a message from the message channel by logical id **fn**. The message will be placed in the buffer **msg**, whose maximum length is **len**. The actual length of the message is returned.

If the incoming message exceeds the buffer length, an error will result.

void pa_clsmsg(int fn);

Closes the message channel by logical id **fn**.

FILE* pa_waitnet(int p, int s);

Waits for a stream connection on the given port **p**. If the **s** flag is enabled, the connection will be encrypted. The file pointer for the connection is returned.

Any number of connections can occur on a single port within the program, but connections to the same port by other programs result in an error.

int pa_waitmsg(int p, int s);

Waits for a message connection on the given port **p**. If the **s** flag is enabled, the connection will be encrypted. The file pointer for the connection is returned.

Any number of connections can occur on a single port within the program, but connections to the same port by other programs result in an error.

int pa_certnet(FILE* f, int which, string cert, int len);

Reads a certificate from the given stream file **f.** The certificate is placed in the buffer **cert**, with length **len**. The actual length of the certificate is returned. Certificates are strings, and may include line feeds to break it up into lines.

int pa_certmsg(int fn, int which, string cert, int len);

Reads a certificate from the given message file **fn**. The certificate is placed in the buffer **cert**, with length **len**. The actual length of the certificate is returned. Certificates are strings, and may include line feeds to break it up into lines.

void pa_certlistnet(FILE *f, int which, pa_certptr* list);

Reads a certificate from the given stream file **f**. The certificate is parsed into a data tree in name/value format, and returned as a pointer to **list**.

The tree is allocated from heap store, and should be recycled when it is no longer needed.

void pa_certlistmsg(int fn, int which, pa_certptr* list);

Reads a certificate from the given message file **fn**. The certificate is parsed into a data tree in name/value format, and returned as a pointer to **list**.

The tree is allocated from heap store, and should be recycled when it is no longer needed.

void pa_certlistfree(pa_certptr* list);

Recycles the given certificate name/value format tree **list**.

14 Option: Command Line Option Processing

Parsing options can be done in a non-OS specific way by the **option** package. It does both the work of accommodating different OS option conventions, as well as providing easy table lookup of options, and processes their parameters.

Presently, option accommodates two different convention for options, that of Unix/Linux/Mac OS X, or Windows:

Format	Operating system
/option	Windows
-o	Unix/Linux/Mac OS X
--option	Unix/Linux/Mac OS X

The option character is set from **pa_optchr()** in services. In all conventions, multiple character options are the normal case, but option will also support single character options if:

- The operating system is Unix/Linux/Mac OS X.
- The single parameter is set on the option call.

The call to parse a series of options is **pa_options(argi, argv, opts, single)**. **argi** is the index of command line words being processed, and **argc** and **argv** are pointer references to the standard C command line arguments. **opts** is a table to look up options, and single indicates that single character options are to be allowed.

Options uses a “transparent” model to parse options. When called, it will check command line words for options, and if found, parse them and skip over any options found. Thus the **pa_options()** call can be made any number of times during processing of command line words, so that options can appear anywhere on the line. Other models are to allow options only at the start of the command line or only at the end.

The option table is of the format:

```
/* option record */
typedef struct {

    string   name; /* name of option */
    int*     flag; /* flag encounter */
    int*     ival; /* integer value */
    float*   fval; /* floating point value */
    string   str;  /* string value */
    int      len;  /* string maximum length */

} pa_optrec, *pa_optptr;
```

The fields are:

Name	The character name of the option, a string.
flag	A pointer to Boolean, if not NULL, the flag will be set if the option is encountered.
ival	A pointer to integer, if not NULL, an integer value is parsed.
fval	A pointer to float, if not NULL, a floating point value is parsed.
str	A pointer to a string buffer, if not NULL, a string value is placed after removing any quotes.
len	The maximum length of the string buffer.

If an option has a parameter associated with it, it can be specified as follows:

--option=42

or

/option:42

(with the leading character matching the operating system in use).

If single character options are in effect, the parameters are gathered from succeeding words:

-ox 42 “hi there”

Option will fill any or all of the option types asked for. For example, if the option was:

--option=42

And all of integer, floating point, and string were specified, then the results would be:

42	Integer.
42.0	Floating point.
“42”	String.

The table ends when the name is specified as NULL.

Putting all this together, an example option use would be (from the program “genwave.c”):

```
int dport = PA_SYNTH_OUT; /* set default synth out */
int freq = 440; /* set default frequency */
int square = FALSE; /* set not square wave */

pa_optrec opttbl[] = {

    { "port",   NULL,    &dport, NULL, NULL, 0 },
    { "p",      NULL,    &dport, NULL, NULL, 0 },
    { "freq",   NULL,    &freq,  NULL, NULL, 0 },
    { "f",      NULL,    &freq,  NULL, NULL, 0 },
    { "square", &square, NULL,   NULL, NULL, 0 },
    { "s",      &square, NULL,   NULL, NULL, 0 },
    { NULL,     NULL,    NULL,   NULL, NULL, 0 }

};

int main(int argc, char **argv)
{

    int argi = 1;

    /* parse user options */
    pa_options(&argi, &argc, argv, opttbl, TRUE);
    if (argc != 1) {

        if (pa_optchr() == '-')
            fprintf(stderr,
                    "Usage: genwave [--port=<port>|-p <port>"
                    "--freq=<freq>|-f <freq>|--square]|-s\n");
        else
            fprintf(stderr,
                    "Usage: genwave [/port=<port>|/p=<port>"
                    "/freq=<freq>|/f <freq>|/square]|/s\n");

        exit(1);

    }

    ...

}
```

In this program the options are **port**, **freq** and **square**. **port** and **freq** are integers, but square is a simple Boolean, we either encountered it or not. Each option has a single character equivalent:

Multicharacter Option	Single Character Option
port	p
freq	f
square	s

and the call to **pa_options()** specifies that Unix/Linux/MAC OS X style single character options are allowed . In Unix/Linux/MAC OS X programs can use the next word in the command line to satisfy single character options with parameters.

There are two general ways that parsing through the **argc/argv** command words in a C program are done. The first is by incrementing the argv array pointer and decrementing the **argc** value. The second is to keep an index number for the **argv** array, and increment that and decrement the **argc** value. The first method is simpler, but keeping an **argi** or **argv** index makes it clear exactly what word we are parsing in the command line. **option** supports this second method.

For string arguments, option will accept quoted strings and automatically remove the quotes when the string (or integer or float) is stored. Thus:

```
--mystring=”This is a long string with spaces”
```

Is possible. **option** will accept either single (‘) or double (“) quotes, as long as they are matched.

The return value for **pa_options()** is 0 if one or more correctly formatted options were found, otherwise 1 if not. If no option was found, the **argi** and **argc** values are not changed. If a list of correct options is followed by one with an error, then the **argi** and **argc** indexes stop at the incorrect option.

Option has another function that does not use **argi** and **argc**, but just takes an option string, **pa_option(s,opts,single)**. Since function is not tied to a **argc**/**argv** word array, it can be used to form your own option parsing system.

Finally, the function **pa_dequote(s)** can be used to remove the quotes from a string. It accepts either single or double quotes, and removes them and puts the string back into the buffer provided on input.

14.1 Functions and Procedures in Option

int pa_options(int* argi, int*argc, char** argv, pa_optrec opts[], int single);

Parse a series of options from an **argc**/**argv** array. **argi** contains the index of the current **argv** array word position. **argc** contains the number of arguments left. **argv** contains the word array. **opts** contains a table of options. **single** is a flag indicating if single character options are to be allowed.

Any of 0 to n options can be parsed. If one or more options is successfully parsed, a 0 is returned, otherwise 1. The integer pointed to by **argi** is incremented for each option successfully parsed, and the integer pointed to by **argc** is decremented. **argv** is not changed.

int pa_option(char* s, pa_optrec opts[], int single);

Parse a single option from a string. **s** contains the string to parse. **opts** contains a table of options. **single** is a flag indicating if single character options are to be allowed.

If the option is successfully parsed, a 0 is returned, otherwise 1.

void pa_dequote(char* s);

Remove quotes from string. Leading and trailing quotes are removed from the provided string. **s** contains the string. Either single (‘) or double (“) quotes can be removed, but leading and trailing quotes must match.

15 Config: The configuration database

There are two ways to pass configuration into a running program:

- Specify options on the command line.
- Use environmental strings inside the program.

Petit-Ami has a third, and far more general method, a configuration database or page. It supports, and is supported by, the program/user/current path method introduced by **services**. It is also the method used by Petit-Ami modules itself. Petit-Ami supports all three methods. The advantages and disadvantages of each are:

Mode	Advantage	Disadvantage
Command line option	Can configure per program run.	Have to repeatedly specify configuration.
Environmental strings	Configuration applies to every run.	Configuration exposed to all programs. Loads up the environment with various program configurations.
Configuration Pages	Configuration applies to every run. Supports per-feature configurations. Supports different configurations for all users, a single user, or a single run instance or directory.	Effort to construct configuration pages.

A configuration page is a file containing a series of “value sets” of the form:

name value

The name can be any name starting with the characters A..Z, a..z and _, and continuing with the characters A..Z, a..z, _ and 0..9. This is the “lookup key” for the value. The value itself can be any character sequence, including spaces, but excluding ‘\n’ (end of line). Thus, the entire series of characters after the name and trailing space are kept as the value. The value can also be blank, containing no characters or only spaces.

Configuration files can also contain blank lines, and a “comment line” of the form:

hi there

Comments can be placed anywhere except in the value field, because they would be added into the value itself (this may or may not be a problem depending on the program that uses the value).

Thus a typical configuration file could be:

```
# my configuration file

myname herman
myipaddr 1.2.3.4
...
```

Etc.

All of your configuration files can be “flat” as shown, but config also supports tree structure in configuration files. For each “branch” of the tree, a **begin** statement appears:

```
begin mybranch

    leaf
    ...

end
```

The meaning of this is that a new sub-catagory “mybranch” is created, and all of the definitions between the **begin** and **end** statements are placed under that branch.

Here is an example for a cut-down version of Petit-Ami’s own configuration file, **petit_ami.cfg**:

```
#
# Configuration file for Petit-ami and submodules
#

#
# Definitions for services module
#
begin services
end

#
# Definitions for terminal mode
#
# Terminal mode applies to several modules, including console and
graph.
#
begin terminal

    #
    # Set default dimensions of main window.
    #
    maxxd 80
    maxyd 25

    #
    # Enable/disable mouse
    #
    mouse 1

    #
    # Enable/disable joystick
    #
    joystick 1

end

#
# Definitions for graphical mode
#
# Graphical mode can apply to multiple modules, but usually is
reserved for just
# one main module.
#
begin graph

    #
    # Send runtime errors to dialog/parent console window
    #
    dialogerr 0
```

```
    #
    # definitions for windows version of graph
    #
    begin windows

        #
        # Definitions for windows diagnostic settings
        #
        begin diagnostics

            #
            # Dump messages (windows messages posted to us)
            #
            dump_messages 0

        end

    end

end
```

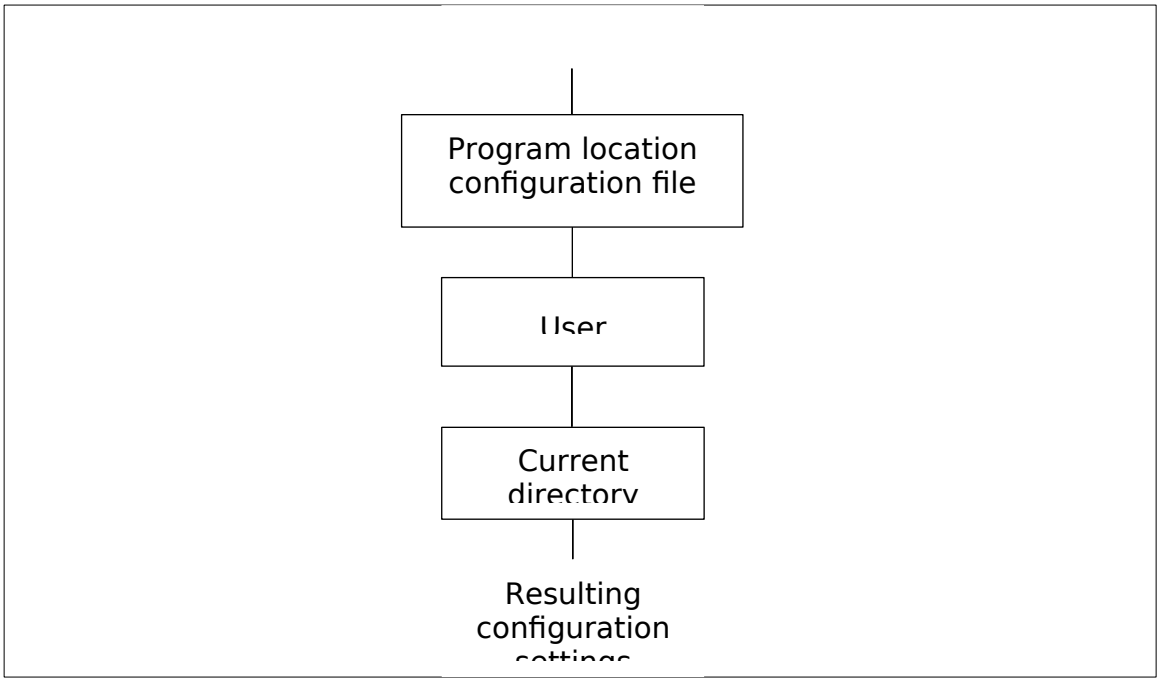
When you parse configuration files, this is done in the series of locations:

- The program location or path.
- The user location or path.
- The current location or path (current directory).

The result of each configuration file parse is a tree of configuration options. Each new tree is merged with the last, with any new definitions replacing the previous definitions, but otherwise each definition is left alone. Thus the resulting tree is the merge of each of the program path, user path and current path trees merged, starting with a merge to a empty list.

The result of this is that the starting basis for the option tree is the option file provided with the program, and available to all users. These definitions are overridden, or replaced, by definitions unique to each user, then possibly overridden by definitions unique to the current directory. Any one (or all) of these configuration files may or may not exist.

Typically the program using the configuration will transfer the definitions it uses to internal configuration parameters. The effect of this is that the default settings for the program exist before being overridden (or not) by each of the program, user and current configuration trees.



Many programs will also provide one or more local options via the command line that can be configured. These will either be configured before or after the configuration tree parse, meaning they may override or be overridden by the configuration pages. Petit-Ami cannot do this because it does not have access to the command line.

For any configuration file(s) a utility exists to parse the file(s) and print the resulting tree:

```
$ prtconfig
Petit-Ami configuration tree:

services              (null)
terminal
    maxxd                 80
    maxyd                 25
    mouse                 1
    joystick              1
console               (null)
graph
    dialogerr             0
    windows
        diagnostics
            dump_messages         0
sound                 (null)
network               (null)
```

This is the configuration tree for Petit-Ami itself.

To read a config tree, use **pa_config(r)** where **r** is a pointer to the root of the tree. It needs to be set null first, or contain a tree to be merged. **pa_config()** looks for the file(s) by he name of **petit_ami.cfg** or **.petit_ami.cfg** (either the visible or invisible file). **pa_config()** looks through the program, user and current path for these files.

To use an alternative filename or search procedure, the function **pa_configfile(fn,r)** is used, where **fn** is the filename (including extension) and **r** is the root, which should be null or contain a starting tree as with **pa_config()**. **pa_configfile()** uses the name as is, complete with path, so it is a building block for your own way of using configuration trees.

To perform your own configuration tree merges, the function **pa_merge(or, nr)** is used, where **or** is the old root, and **nr** is the new root. As before, the new tree definitions replace the old tree definitions of the same name and branch(s).

To find definitions in the resulting configuration trees, the function **pa_schlst(id,r)** is used, where **id** contains the key name to be found, and **r** contains the tree. It returns a pointer to the value structure if found, or NULL if none is found.

In practice, this simple search function works well even if multiple levels of nesting are present. An example of this is the code from the Windows graphics module:

```
pa_valptr config_root; /* root for config block */
pa_valptr term_root; /* root for terminal block */
pa_valptr graph_root; /* root for graphics block */
pa_valptr diag_root; /* root for diagnostics block */
pa_valptr win_root; /* root for windows */
pa_valptr vp;

/* get setup configuration */
config_root = NULL;
pa_config(&config_root);

/* find "terminal" block */
term_root = pa_schlst("terminal", config_root);
if (term_root && term_root->sublist) term_root = term_root->sublist;

/* find x an y max if they exist */
vp = pa_schlst("maxxd", term_root);
if (vp) maxxd = strtol(vp->value, &errstr, 10);
if (*errstr) error(ecfgval);
vp = pa_schlst("maxyd", term_root);
if (vp) maxyd = strtol(vp->value, &errstr, 10);
if (*errstr) error(ecfgval);

/* find graph block */
graph_root = pa_schlst("graph", config_root);
if (graph_root) {

    vp = pa_schlst("dialogerr", graph_root->sublist);
    if (vp) dialogerr = strtol(vp->value, &errstr, 10);
    if (*errstr) error(ecfgval);

    /* find windows subsection */
    win_root = pa_schlst("windows", graph_root->sublist);
    if (win_root) {

        /* find diagnostic subsection */
        diag_root = pa_schlst("diagnostics", win_root->sublist);
        if (diag_root) {

            vp = pa_schlst("dump_messages", diag_root->sublist);
            if (vp) dmpmsg = strtol(vp->value, &errstr, 10);
            if (*errstr) error(ecfgval);

        }

    }

}
```

The code works on the idea that the **pa_schlst()** routine can find either value entries or branch entries, and once a branch is found, searching within that sublist is easy.

To print out configuration trees of any depth, use **pa_prttre(l)**, where **l** is the configuration tree root. This prints a tree structured list of entries as shown before with **pa_prtconfig()**.

16 Sound module plugins

The sound module has the ability to accept plug-ins. This was necessitated by the fact that Linux does not have a standard midi synthesizer built into the operating system. However, the sound plug-ins are a general purpose facility, and the plan is to extend the ability to other operating systems. See “Linux details” for more on this.

The sound plug-in facility gives external modules the ability to declare pseudo-devices, and intercept I/O calls, for any of:

- Synthesizer/MIDI input.
- Synthesizer/MIDI output.
- PCM wave input.
- PCM wave output.

A different call is provided for each:

_pa_synthoutplug()	For a synthesizer output pseudo-device.
_pa_synthinplug()	For a synthesizer input pseudo-device.
_pa_waveoutplug()	For a PCM wave output pseudo-device.
_pa_waveinplug()	For a PCM wave input pseudo-device.

A plug-in wishing to expose a pseudo-device gives the name of the device, and a series of vectors to the routines that will handle calls for that device. The call registers the name of the device and assigns it a device number. Any calls using that device will then be re-routed to the plug-in code.

In addition to the I/O functions for the plug-in device, a plug-in has the ability to both read and write string data parameters. These are values specific to the pseudo-device being implemented by the plug-in. A plug-in has two ways to have parameters within it set. The first way is to parse configuration strings (see 15 ”

Config: The configuration database"). The second is to implement read and write data parameters via the plug-in interface. The first method gives the user the ability to configure the plug-in. The second gives a program using sound the ability to configure the plug-in.

One example of a pseudo-device given above is a software synthesizer. Another that is provided with Petit-Ami is a MIDI dumper that can decode and print all midi messages that pass through the device[4]. Other possibilities include MIDI recorders, PCM wave echo or other effects generators.

Operating systems often provide their own sound plug-in facilities. Further, some programs such as DAWs (Digital Audio Workstations) may have their own plug-in format. The main advantage of using Petit-Amis plug-in format is that it is portable to anywhere Petit-Ami goes.

1.2 Functions in sound plug-ins

void _pa_synthoutplug(int addend, string name, void (*opnseq)(int p),
void (*clsseq)(int p), void (*rdseq)(int p, pa_seqptr sp),
int (*setparam)(int p, string name, string value),
void (*getparam)(int p, string name, string value, int l));

Declares a synthesizer/MIDI output pseudo-device. The **addend** Boolean indicates if the device is to be inserted at the front or the end of the device list. If true, it is inserted at the end, otherwise at the front.

The **opnseq(p)** vector will be called with the device number as parameter **p** when the device is opened, and the **clsseq(p)** vector will be called with the device number as parameter **p** when the device is closed. The device number may be ignored, or it may be used to differentiate between multiple devices if the same routine is used for the different devices.

The **wrseq(p, sp)** vector is called with a MIDI message structure **pa_seqmsg** pointer **sp**. The next MIDI message is written from the the structure to the pseudo-device **p**.

The vector **setparm(p, n, v)** sends a string value to device **p** with name **n**, and value **v** to the pseudo device. The set of values the pseudo-device implements is specific to the pseudo-device.

The vector **getparam(p, n, v, l)** gets a string from device **p** with name **n**, and value **v** with length **l**. The buffer **v** is filled with the resulting string. The set of values the pseudo-device implements is specific to the pseudo-device.

void _pa_synthinplug(int addend, string name, void (*opnseq)(int p), void (*clsseq)(int p),
void (*wrseq)(int p, pa_seqptr sp), int (*setparam)(int p, string name, string value),
void (*getparam)(int p, string name, string value, int l));

Declares a synthesizer/MIDI input pseudo-device. The **addend** Boolean indicates if the device is to be inserted at the front or the end of the device list. If true, it is inserted at the end, otherwise at the front.

The **opnseq(p)** vector will be called with the device number as parameter **p** when the device is opened, and the **clsseq(p)** vector will be called with the device number as parameter **p** when the device is closed. The device number may be ignored, or it may be used to differentiate between multiple devices if the same routine is used for the different devices.

[4] Although the dump plug-in is operating system independent, it is included in the Linux section since that is the only system that implements plug-ins at this time.

The **rdseq(p, sp)** vector is called with a MIDI message structure **pa_seqmsg** pointer **sp**. The next MIDI message is read into the the structure by the pseudo-device **p**.

The vector **setparm(p, n, v)** sends a string value to device **p** with name **n**, and value **v** to the pseudo device. The set of values the pseudo-device implements is specific to the pseudo-device.

The vector **getparam(p, n, v, l)** gets a string from device **p** with name **n**, and value **v** with length **l**. The buffer **v** is filled with the resulting string. The set of values the pseudo-device implements is specific to the pseudo-device.

void _pa_waveoutplug(int addend, string name, void (*open)(int p),
void (*close)(int p), void (*chanwavout)(int p, int c),
void (*ratewavout)(int p, int r), void (*lenwavout)(int p, int l),void (*sgnwavout)(int p, int s),
void (*fltwavout)(int p, int f), void (*endwaveout)(int p, int e),
void (*wrwav)(int p, byte* buff, int len), int (*setparam)(int p, string name, string value),
void (*getparam)(int p, string name, string value, int l));

Declares a PCM wave output pseudo-device. The **addend** Boolean indicates if the device is to be inserted at the front or the end of the device list. If true, it is inserted at the end, otherwise at the front.

The **opnseq(p)** vector will be called with the device number as parameter **p** when the device is opened, and the **clsseq(p)** vector will be called with the device number as parameter **p** when the device is closed. The device number may be ignored, or it may be used to differentiate between multiple devices if the same routine is used for the different devices.

The **wrwav(p, b, l)** vector is called with a buffer containing wave samples **b** with length **l**. The wave samples are written to the pseudo-device.

The vector **setparm(p, n, v)** sends a string value to device **p** with name **n**, and value **v** to the pseudo device. The set of values the pseudo-device implements is specific to the pseudo-device.

The vector **getparam(p, n, v, l)** gets a string from device **p** with name **n**, and value **v** with length **l**. The buffer **v** is filled with the resulting string. The set of values the pseudo-device implements is specific to the pseudo-device.

void _pa_waveinplug(int addend, string name, void (*open)(int p), void (*close)(int p),
int (*chanwavin)(int p), int (*ratewavin)(int p),
int (*lenwavin)(int p), int (*sgnwavin)(int p),
int (*fltwavin)(int p), int (*endwavein)(int p), int (*rdwav)(int p, byte* buff, int len),
int (*setparam)(int p, string name, string value),
void (*getparam)(int p, string name, string value, int l));

Declares a PCM wave input pseudo-device. The **addend** Boolean indicates if the device is to be inserted at the front or the end of the device list. If true, it is inserted at the end, otherwise at the front.

The **opnseq(p)** vector will be called with the device number as parameter **p** when the device is opened, and the **clsseq(p)** vector will be called with the device number as parameter **p** when the device is closed. The device number may be ignored, or it may be used to differentiate between multiple devices if the same routine is used for the different devices.

The **rdwav(p, b, l)** vector is called with a buffer for wave samples **b** with length **l**. The wave samples are read from the pseudo-device.

The **rdwav(p, sp)** vector is called with a MIDI message structure **pa_seqmsg** pointer **sp**. The next MIDI message is read into the the structure by the pseudo-device **p**.

The vector **setparm(p, n, v)** sends a string value to device **p** with name **n**, and value **v** to the pseudo device. The set of values the pseudo-device implements is specific to the pseudo-device.

The vector **getparam(p, n, v, l)** gets a string from device **p** with name **n**, and value **v** with length **l**. The buffer **v** is filled with the resulting string. The set of values the pseudo-device implements is specific to the pseudo-device.

17 Example Applications

graph_games	Example graphical games.
breakout.c	Break the brick wall game.
pong.c	Pong handball game.
graph_programs	Example graphical programs.
ball1.c	“dazzlers” using balls.
ball2.c	“”
ball3.c	“”
ball4.c	“”
ball5.c	“”
ball6.c	“”
clock.c	A resizable clock program.
line1.c	“dazzlers” using lines.
line2.c	“”
line4.c	“”
line5.c	“”
pixel.c	Pixel set demo.
network_programs	Network example programs.
getmail.c	Get mail from server.
getpage.c	Get page from http[s] server.
gettys.c	Example server.
msgclient.c	Example message client.
msgserver.c	Example message server.
prtcertmsg.c	Print ssl certificate for messages.
prtcertnet.c	Print ssl certificate for TCP/IP.
sound_programs	Example sound programs.
connectmidi.c	Connect MIDI input to output.

connectwave.c	Connect PCM wave input to output.
genwave.c	Generate sine/square wave.
keyboard.c	Use keyboard as MIDI controller.
play.c	Play songs in MS Basic format.
playmidi.c	Play MIDI file.
playwave.c	Play PCM wave file.
printdev.c	Print sound devices.
random.c	Play random notes.
terminal_games	Example terminal games.
mine.c	Mine sweeper game.
pong.c	Pong game.
snake.c	Snake game.
terminal_programs	Example terminal programs.
editor.c	Text editor.
editor.not	Text editor notes.
wator.c	Wator visual demo.
tests	Test programs.
event.c	Event display.

18 Libc and alternatives

1.3 stdio

stdio.c is an implementation of the standard I/O functions:

```
FILE *fopen(const char* filename, const char* mode);
FILE *freopen(const char* filename, const char* mode, FILE* stream);
FILE *fdopen(int fd, const char *mode);
int fflush(FILE* stream);
int fclose(FILE* stream);
int remove(const char *filename);
int rename(const char *oldname, const char *newname);
FILE *tmpfile(void);
char *tmpnam(char s[]);
int setvbuf(FILE* stream, char *buf, int mode, size_t size);
void setbuf(FILE* stream, char *buf);
int fprintf(FILE* stream, const char *format, ...);
int printf(const char* format, ...);
int sprintf(char* s, const char *format, ...);
int vprintf(const char* format, va_list arg);
int vfprintf(FILE* stream, const char *format, va_list arg);
int vsprintf(char* s, const char *format, va_list arg);
int fscanf(FILE* stream, const char *format, ...);
int scanf(const char* format, ...);
int sscanf(const char* s, const char *format, ...);
int fgetc(FILE *stream);
int getc(FILE *stream);
char *fgets(char *s, int n, FILE *stream);
int fputc(int c, FILE *stream);
int fputs(const char *s, FILE *stream);
int putc(int c, FILE *stream);
int getchar(void);
char *gets(char *s);
int putc(int c, FILE *stream);
int putchar(int c);
int puts(const char *s);
int ungetc(int c, FILE *stream);
size_t fread(void *ptr, size_t size, size_t nobj, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE
*stream);
int fseek(FILE* stream, long offset, int origin);
long ftell(FILE* stream);
void rewind(FILE* stream);
int fgetpos(FILE* stream, fpos_t *ptr);
int fsetpos(FILE* stream, const fpos_t *ptr);
void clearerr(FILE* stream);
int feof(FILE* stream);
int ferror(FILE* stream);
void perror(const char *s);
```

```
int fileno(FILE* stream);
```

implemented local to Petit-Ami. The purpose of this is that those calls can be redirected to be presented as a stream within the Peitit-Ami package. This is done for several reasons:

- The text is presented as a screen surface in Terminal.
- The text is presented in a graphical surface in Graphics.
- The I/O text is sent via a TCP/IP connection.

The stdio calls are implemented compatibly with standard ANSI C stdio calls. The difference is that the lowest level I/O that goes through the calls:

```
static ssize_t read(int fd, void* buff, size_t count);
static ssize_t write(int fd, const void* buff, size_t count);
static int open(const char* pathname, int flags, int perm);
static int close(int fd);
static int unlink(const char* pathname);
static off_t lseek(int fd, off_t offset, int whence);
```

can be overridden by the calls:

```
void ovr_read(vt_read_t nfp, vt_read_t* ofp) { *ofp = vt_read; vt_read = nfp; }
void ovr_write(vt_write_t nfp, vt_write_t* ofp) { *ofp = vt_write; vt_write = nfp; }
void ovr_open(vt_open_t nfp, vt_open_t* ofp) { *ofp = vt_open; vt_open = nfp; }
void ovr_close(vt_close_t nfp, vt_close_t* ofp) { *ofp = vt_close; vt_close = nfp; }
void ovr_unlink(vt_unlink_t nfp, vt_unlink_t* ofp)
      { *ofp = vt_unlink; vt_unlink = nfp; }
void ovr_lseek(vt_lseek_t nfp, vt_lseek_t* ofp) { *ofp = vt_lseek; vt_lseek = nfp; }
```

Which allows other modules (or even your modules) to redirect I/O, perhaps selectively, to other purposes. Each of the overriders places their own calls into vectors in the Petit-Ami stdio package, and gives the caller back a vector to the old function. This means the overrider can perform the action itself, send it back to the original call, or modify the data and then send it on. This is a classic override in the sense of object oriented languages.

1.4 Linker overriding

Using either gcc or llvmgcc, when stdio is properly linked to the target program, the Petit-Ami stdio module replaces the functions in stdio from the ANSI C layer in the target operating system. For Linux this is glibc. For windows it is msvcrt. Thus Petit-Ami does not have to replace the entire libc collection of functions, but only the code within stdio. This works for all of Windows and Mac OS X, and only Linux is different.

1.5 The glibc override library

glibc is the implementation of libc used in Linux. Unfortunately, Petit-Ami's stdio implementation does not work correctly in the Linux environment. It does replace the stdio part of glibc correctly. The issue is that several other modules that are included with the normal tool chain in Linux use the stdio FILE structure, which is supposed to be opaque (having a structure unknown to the caller) and assume it has the fields given it by glibc, and then directly manipulate them. Among the programs that do this is gdb. Only Linux programs do this, because only Linux programs use glibc.

There were two ways to deal with this:

- Match the structure of glibc FILE within Petit-Ami.
- Create a version of glibc that has override calls.

The drawback of the first solution is that there is always another program/module that can "reach around" the FILE data structure and do unforeseen things to it, wrecking upward compatability. The drawback of the second solution is that all of the baggage of glibc must be carried along with Petit-Ami, and also that the glibc that is built must match the Linux kernel in use.

Petit-Ami uses a custom build of glibc at present. This gives maximum compatibility with existing builds. Nothing can emulate glibc like glibc. It does, however, make the preparation of a glibc module more difficult.

1.5.1 Selecting a precompiled glibc

First of all, the only result of glibc is the final module it produces, libc.so or libc.a. I have produced several of these for the latest Linux Kernal versions and placed them in bin/glibc. There is a symbolic link to the proper libc.so or libc.a as libc.so.6 or libc.a[5]. These links are setup by the script:

```
setver_glibc [<version>]
```

If you don't specify a version, the script will automatically determine what your kernel version is, and set up the symbolic links to match that. If there is no precompiled glibc for your platform, you will get an error:

*** libc override file by version <version> does not exist

In this case, you need to build your own glibc.

If you do specify a version, it will select that version to link to. This is only useful when cross compiling for another system, since the programs generated will not run on any kernel but the present one.

There is also a script to simply print the current glibc version installed on your system:

```
ver_glibc
```

1.5.2 Structure of the glibc build system

glibc is kept in the Petit-Ami directory glibc. To make obtaining the proper glibc and building it as simple as possible, a few scripts are included in the bin directory that automate the process.

1.5.3 Compiling a new glibc

The script:

```
build_glibc [<version>]
```

Exists to do the work of fetching the proper glibc version from the sourceware.org git repository, selecting the correct version of glibc, patching and building it, and putting the result into the bin/glibc directory. You can then select it with setver_glibc as above.

[5] static linking for glibc does not work at this writing.

If you don’t give build_glibc a version number argument, it will find the current glibc version installed on your system, and fetch and build that. Otherwise, you may specify a particular version to build. Thus you can build any number of versions, and use setver_glibc to select the one you want.

To build a particular glibc, you need the files:

glibc_x.xx_override.patch

And

libc_x.xx_.map.patch

In the petit_ami/glibc directory, where x.xx matches the kernel version to be used. These files supply the override code, and the override call header definitions, respectively.

Athough the patch files must be matched to the version, at this writing the major change occurred between the 2.27 and 2.28 versions. Later versions of the files are simply copies of each other, and you can copy the existing files to create a new set of patch files.

1.5.4 The fix: a new version of stdio

I expect at some point to implement the first solution to the Linux problem, that of compensating for the expected private fields of the glibc FILE structure. This should make working with Linux builds much easier. However, the patched version of glibc will always be available, since in the future some new system module could break Linux builds as outlined above.

19 Directory layout

The directory format for the Petit-Ami project generally follows the standard for GNU project spaces.

```
bin                                 Binaries (executables)
      glibc                         glibc build/patch area.
            libc-ovr-2.26.a         Static version of override glibc.
            libc-ovr-2.26.so        Dynamic version of override glibc.
            libc-ovr-2.27.a         “”
            libc-ovr-2.27.so        “”
            libc-ovr-2.29.a         “”
            libc-ovr-2.29.so        “”
            libc-ovr-2.30.a         “”
            libc-ovr-2.30.so        “”
      build_glibc                   Script to fetch and build glibc.
      create_glibc_override_patch   Script to generate glibc override
      patches.
      gencert                       Script to generate security certificate.
      libc.so.6                     Symbolic link to current version of dy-
                                    namic override glibc.
      libc.a                        Symbolic link to current vesion of
                                    static override glibc.
      Makecerts                     Script to generate certificate set.
      setver_glibc                        Set version of glibc to current/
      given.
      ver_glibc                     Show current version of glibc.

cpp                                 C++ program files.
      cpp.not
      terminal.cpp
doc                                 Documentation.
      petit_ami.docx
glibc                               The glibc build/patch area.
      glibc_2.27_override.patch
      glibc_2.28_override.patch
      glibc_2.29_override.patch
      libc_2.27_.map.patch
      libc_2.28_.map.patch
graph_games                         Example graphical games.
      breakout.c
      pong.c
graph_programs                      Example graphical programs.
      ball1.c                       “dazzlers” using balls.
      ball2.c                       “”
      ball3.c                       “”
      ball4.c                       “”
      ball5.c                       “”
      ball6.c                       “”
      car_rev.wav                   Used by ball6.
      clock.c                       A resizable clock program.
      line1.c                       “dazzlers” using lines.
      line2.c                       “”
      line4.c                       “”
```

```
      line5.c                    ""
      pixel.c                    Pixel set demo.
      pong.wav                   Used by ball6.
hpp                              C++ header files.
      terminal.hpp               Header file for C++ wrapper.
include                          Include files for C.
      config.h                   config package header.
      graphics.h                 graphics package header.
      localdefs.h                commonly used defines in Petit-Ami.
      network.h                  network package header.
      option.h                   Command line options package header.
      services.h                 services package header.
      sound.h                    sound package header.
      terminal.h                 terminal package header.
libc                             The libc source for Petit-Ami.
      hooktest.c                 Test the override function in stdio.
      stdio.c                    Petit-Ami specific stdio with overrides.
      stdio.h                    Petit-Ami specific stdio header.
linux                            Linux specific source files.
      dumpsynthplug.c            MIDI dump plug-in for sound.
      fluidsynthplug.c           Fluidsynth wave table plug-in software synthe-
sizer.
      graphics.c                 graphics module.
      graphics.not               graphics module notes.
      network.c                  network module.
      network.not                network module notes.
      services.c                 services module.
      services.not               services module notes.
      sound.c                    sound module.
      sound.not                  sound module notes.
      terminal.c                 terminal module.
      terminal.not               terminal module notes.
mid                              Midi example files.
      bach_846_format0.mid
network_programs                 Network example programs.
      getmail.c                  Get mail from server.
      getpage.c                  Get page from http[s] server.
      gettys.c                   Example server.
      msgclient.c                Example message client.
      msgserver.c                Example message server.
      prtcertmsg.c               Print ssl certificate for messages.
      prtcertnet.c               Print ssl certificate for TCP/IP.
sound_programs                   Example sound programs.
      connectmidi.c              Connect MIDI input to output.
      connectwave.c              Connect PCM wave input to output.
      genwave.c                  Generate sine/square wave.
      keyboard.c                 Use keyboard as MIDI controller.
      play.c                     Play songs in MS Basic format.
      playmidi.c                 Play MIDI file.
      playwave.c                 Play PCM wave file.
      printdev.c                 Print sound devices.
      random.c                   Play random notes.
```

```
stub                        No-operation API code for Petit-Ami.
      graphics.c            graphics module.
      keeper.c              .so “keeper” code.
      network.c             network module.
      services.c            services module.
      sound.c               sound module.
      terminal.c            terminal module.
terminal_games              Example terminal games.
      mine.c                      Mine sweeper game.
      pong.c                      Pong game.
      snake.c               Snake game.
terminal_programs           Example terminal programs.
      editor.c              Text editor.
      editor.not            Text editor notes.
      wator.c               Wator visual demo.
tests                       Test programs.
      event.c               Event display.
      graphics_test.c       graphics module test.
      management_test.c     Window management test.
      mypic.bmp             Test data files.
      mypic.jpg             “”
      mypic1.bmp            “”
      mypic1.jpg            “”
      scnsize.txt           Screen dimensions test file.
      services_test.c       services module test.
      services_test1.c      Part of services module test.
      sound_test.c                sound module test.
      term.c                      Terminal emulator.
      terminal_test.c       terminal module test.
      widget_test.c               Widgets test.
utils                       Portable utilities.
wav                         Sound sample files.
      BabyElephantWalk60.wav
windows                     Windows specific source files.
      graphics.c            graphics module.
      graphics.not                graphics module test.
      network.c             network module.
      network.not           network module notes.
      services.c            services module.
      sound.c               sound module.
      sound.not             sound module notes.
      terminal.c            erminal module.
      terminal.not                terminal module notes.
configure                   Linux configuration script.
configure.bat                     Windows configuration script.
INSTALL                     Instructions to install Petit-Ami.
LICENSE                     Petit-Ami BSD license details.
Makefile                    Project make file.
NEWS                        News on current version.
petit_ami.cfg                     Configuration database.
README                            Project overview/introduction.
setpath                     Linux add local bin to path.
```

```
setpath.bat                    Windows add local bin to path.
TESTREPORT                     Report on test coverage for current version.
TODO                           List of needed improvements.
```

20 Installing and building Petit-Ami

Petit-Ami follows the GNU guidelines for configure and build steps. Steps to install:

Linux:

```
. setpath [recommended]
./configure
make
```

Windows:

```
setpath [recommended]
configure
make
```

If you get something like:

```
Version number: 2.31
*** libc override file by version 2.31 does not exist
```

Then you have to build a new version of glibc. See 17.3 “The glibc override library”.

The make step makes all Petit-Ami libraries and target programs. If you want to only build a particular program or component, execute:

```
make <program/module>
```

1.6 Standard make targets

all Makes all libraries and target programs.

clean Removes all binaries and intermediate files.

21 Testing Petit-Ami

The tests for Petit-Ami are kept in the test directory. The tests will be made along with the other programs and libraries in the Makefile. Besides the main tests, it is also customary to run through the example programs during a test series.

The complete test series performed, as well as a report of what tests were last completed, what the results were (PASS/FAIL), as well as when they were last performed and on what version, can be found in the text file TESTREPORT.

22 Windows Specific Details

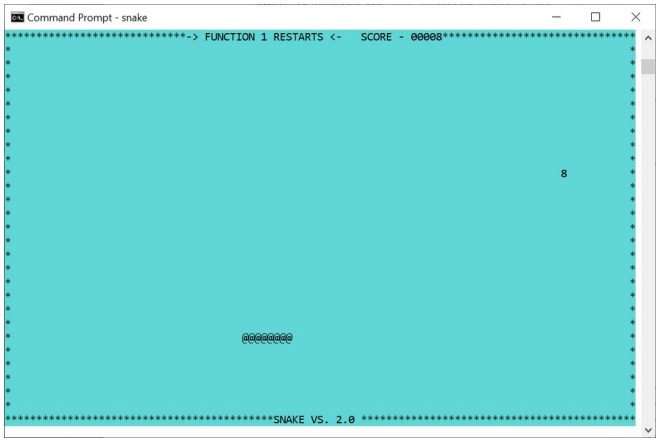
1.7 Terminal

The **terminal** module in Windows is implemented with the Windows Console API. By default, **terminal** takes the display and buffer sizes that exist when **terminal** starts.

1.7.1 Transparent mode

Windows console has two surfaces of importance, the current buffer size and the current display window. The display window is a subset of the buffer, and it moves down as lines are printed. The text scrolled off the top of the display region stay in the buffer, acting as a "history" of printed lines.

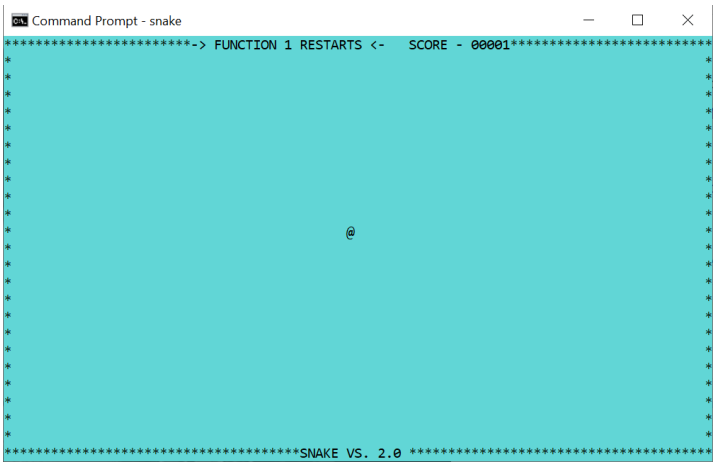
Windows **terminal** has an ability I call "transparent mode". If the selected screen is left at the default 1, and no control features executed (reposition cursor, etc.), then the console window will behave just as if the program did not use the **terminal** module. Scroll down will behave as normal. A program that takes advantage of cursor positioning will operate in a fixed window defined by the current display region:



A console program presenting in the middle of the buffer.

A program like "play" (in the sound_programs directory) does not do any display manipulation, but only uses the **terminal** module for its input event capability and timers. It does not flip the display screen. Thus it appears to function as a console line oriented program, even while it processes events.

A **terminal** program that does move to a different screen:



A console program using a new buffer.

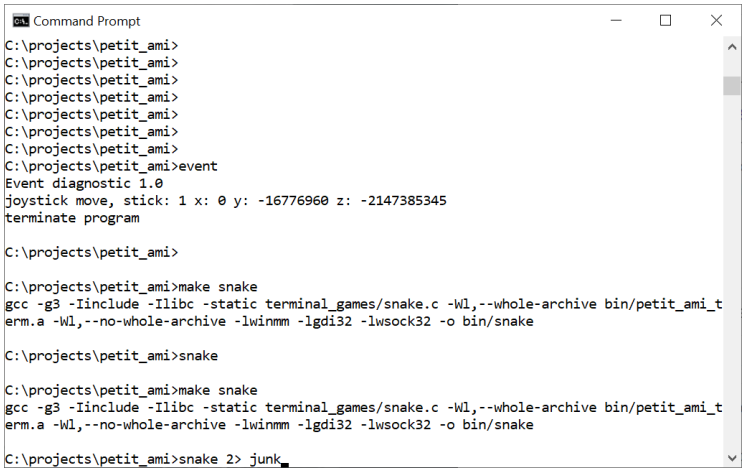
Gets a buffer that is the same as the display area. If it restores the screen to 1,1, as is the custom with Petit-Ami programs, the old buffer, display area, and screen contents will be restored.

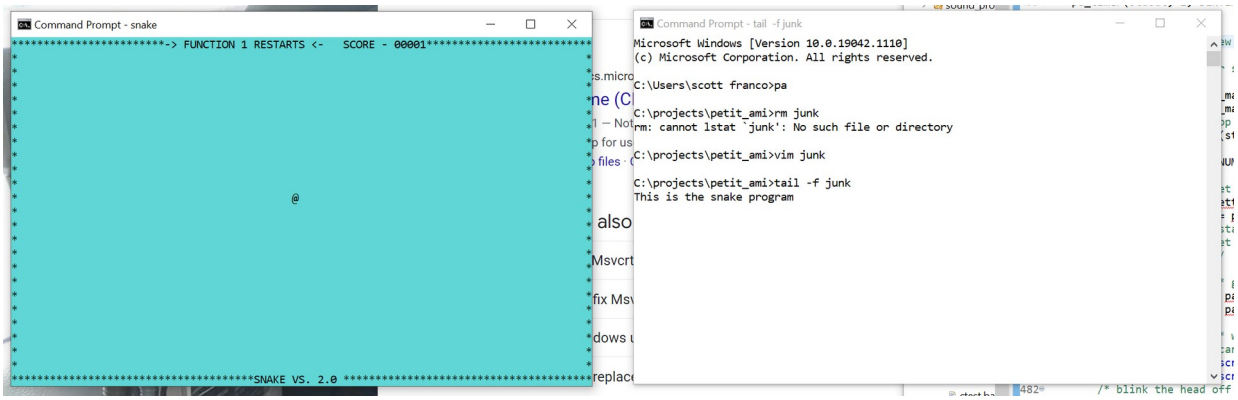
1.7.2 Configuration settings

At this writing, Windows terminal does not implement any configuration settings.

1.7.3 Redirected files

The files **stdin** and **stdout** are redirected to the terminal surface. Any other files, including the **stderr** file, bypass the terminal module and print directly. This can cause the terminal display module to malfunction. I recommend instead that you reroute the messages from **stderr** to another window:





Using a program such as “tail” from the Mingw toolset. This gives you a good debug method without interfering with the main display.

1.7.4 Bypass input

If the input from **stdin** is directly read, as opposed to using the **pa_event()** call, terminal will automatically perform echoing of all input characters, and will buffer the characters up into lines. The only editing of the line currently performed is to erase characters to the left with the backspace key.

1.8 Graphics

The graphics module is implemented with the Win32 API. It comes up by default in 80x25 character mode. The font used for the default PA_FONT_TERM is the system fixed font. The main (default) window will be opened as a separate window, even if it were started from a console window.

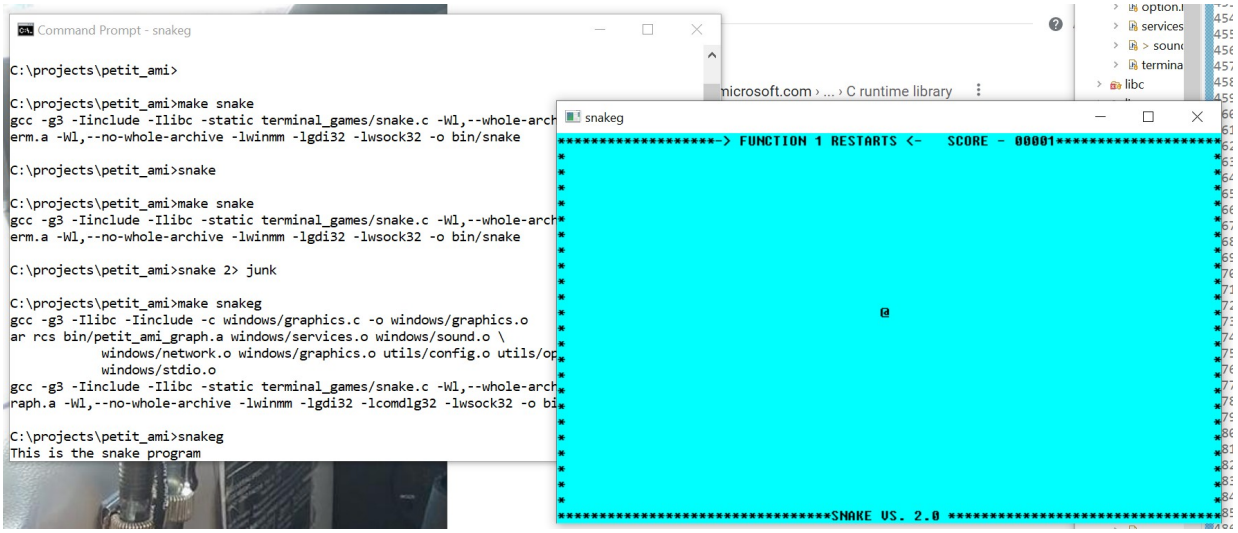
1.8.1 Configuration settings

graphics inplements several configuration settings:

Section	Name	Function
terminal	maxxd	Sets the default character width of the main window.
terminal	maxyd	Sets the default line height of the main window.
terminal	joystick	Enables (1) or disables (0) any joystick attached.
terminal	mouse	Enables (1) or disables (0) any mouse attached.
terminal	dump_event	Dumps petit_ami event codes to stderr. 1 enables, 0 disables.
graphics	dialogerr	Sends all errors to a dialog. Default is stderr. 1 enables, 0 disables.
Graphics/ windows/ diagnostics	dump_messages	Dumps windows messages to stderr. 1 enables, 0 disables.

1.8.2 Redirected files

The **stdin** and **stdout** are redirected to the main window. All other files, including stderr, are left to their default stream locations. This can be useful. For example all **stderr** prints from a program started in a console window print to the console window:

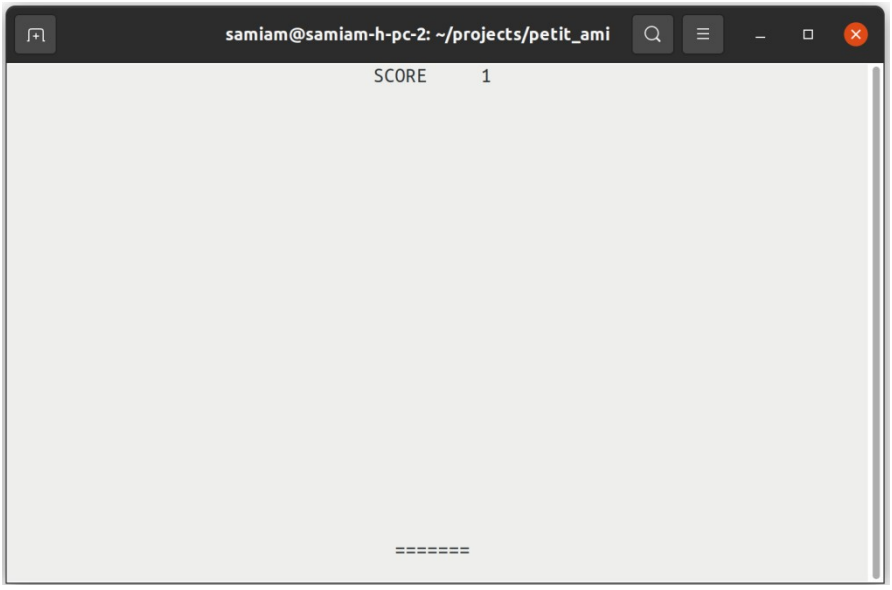


1.8.3 Bypass input

As with **terminal**, any input from **stdin** that is read directly gets echoed and line buffered.

23 Linux Specific details

1.9 Terminal



A Linux terminal program (xterm).

Linux uses XTERM serial control codes to function. XTERM is based on the VT terminal series from DEC (Digital Equipment Corporation), which were based on ANSI X3.64 (ISO 6429) standardized terminal control codes. XTERM supports codes from a series of VT terminals, from the VT102 onward. It also supports additional control codes to allow things such as remote operation of a mouse.

Linux terminal uses the display size it is given on startup.

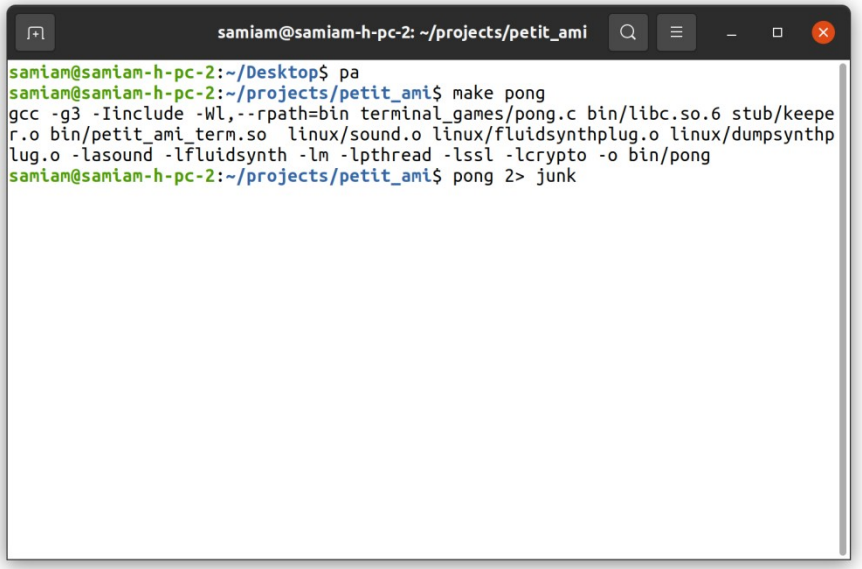
At this writing, XTERM is not capable of supporting transparent mode. Instead, terminal always flips the screen buffer to a new, blank screen and the original XTERM screen is preserved. When terminal exits, the original screen is restored.

1.9.1 Configuration settings

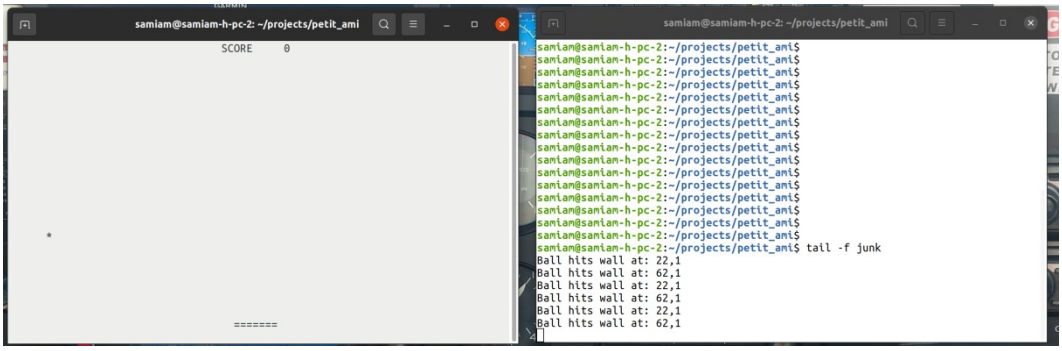
At this writing, Windows terminal does not implement any configuration settings.

1.9.2 Redirected files

The files **stdin** and **stdout** are redirected to the terminal surface. Any other files, including the **stderr** file, bypass the terminal module and print directly. This can cause the terminal display module to malfunction. I recommend instead that you reroute the messages from stderr to another window:



Redirecting stderr to file “junk”.



Now prints to stderr go to a separate window.

Using a program such as “tail”. This gives you a good debug method without interfering with the main display.

1.9.3 Bypass input

If the input from **stdin** is directly read, as opposed to using the **pa_event()** call, terminal will automatically perform echoing of all input characters, and will buffer the characters up into lines. The only editing of the line currently performed is to erase characters to the left with the backspace key.

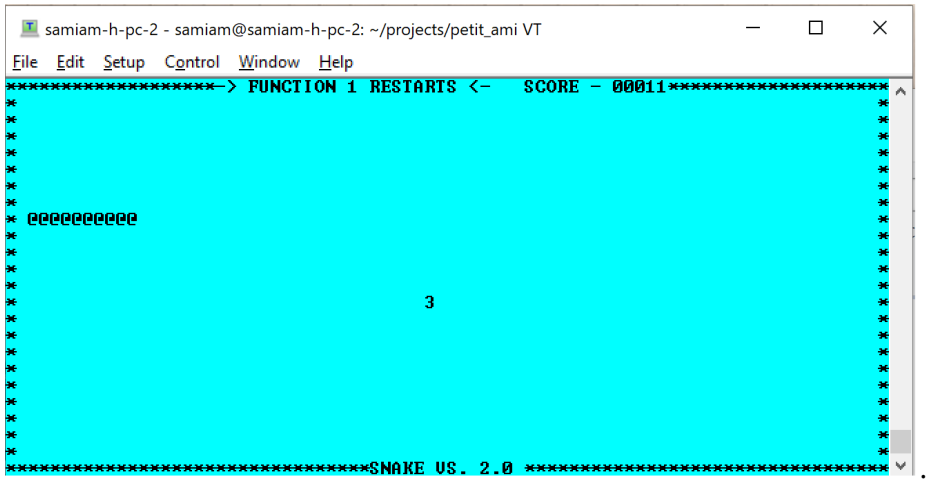
1.9.4 Remote operation

xterm control codes are widely emulated. Even the remote operation of a mouse can function on a remote computer. Natually a ssh/ssl link that does not interfere with the control characters it is streaming can carry a remote connection. No protocol exists at this time to remote operate the joystick.

The following systems have been found to work remotely:

- Linux ssh with xterm, mouse supported.
- Windows with correct ssh in command shell, mouse not supported.
- Windows Teraterm, mouse supported.

Of course, the XWindow protocol can always be exported remotely to run any graphics window remotely, including xterm



An xterm session running on Windows TeraTerm, but hosted on a remote Linux system.

1.10 Graphics

The Linux graphics module uses the XWindows API to present graphics. It sizes the main window by default to 80x25 characters, and sizes the console font to 18 points high. It uses the “bitstream: courier 10 pitch” font by default for the terminal font. At present, only the Xft fonts are used, and only uses scalable fonts.

1.10.1 Configuration settings

graphics inplements several configuration settings:

Section	Name	Function
terminal	maxxd	Sets the default character width of the main window.
terminal	maxyd	Sets the default line height of the main window.
terminal	joystick	Enables (1) or disables (0) any joystick attached.
terminal	mouse	Enables (1) or disables (0) any mouse attached.
terminal	dump_event	Dumps petit_ami event codes to stderr. 1 enables, 0 disables.

graphics	dialogerr	Sends all errors to a dialog. Default is stderr. 1 enables, 0 disables.
Graphics/ xwindow/ diagnostics	dump_messages	Dumps windows messages to stderr. 1 enables, 0 disables.
Graphics/ xwindow/ diagnostics	Print_font_metrics	Prints the font metrics of the terminal type font.

1.10.2 Redirected files

The **stdin** and **stdout** are redirected to the main window. All other files, including stderr, are left to their default stream locations. This can be useful. For example all stderr prints from a program started in a xterm window print to the xterm window:

1.10.3 Bypass input

As with terminal, any input from **stdin** that is read directly gets echoed and line buffered.

1.11 Sound

The sound module implements the plug-in standard (16 ”Sound module plugins”). The reason for this is that under Linux, even though the ALSA interface supports a plug in synthesizer, that is not set up by default in most or all distributions, and having it defined as a plug-in means your program can ship with a synthesizer installed with it. At this writing, I included a plug-in for Fluidsynth, a wave table based synthesizer that uses the Sound Fonts format. The default sound font is:

/usr/share/sounds/sf2/FluidR3_GM.sf2

The Fluidsynth plug-in creates 4 instances of the synthesizer by default, and these will appear in the front of the MIDI synthesizer output device table.

The Fluidsynth plug-in does not take any parameter sets at this writing.

Sound also has a “dump” plug-in. This plug-in will print a decoded dump of any MIDI messages that pass through it, and then send the MIDI messages on to another input port. This is done by a parameter set:

connect <input port>

Where <input port> is another MIDI input port.

1.11.1 ALSA device names

For the most part, sound takes the device names given by ALSA. The exception would be for **sound** module plug-ins. The name of the device also includes details about the device itself. The important thing when selecting a device is the device number and kind (MIDI in/out, wave in/out). This unambiguously selects the device.

Care must be taken when selecting an input wave device. ALSA provides a lot of automatic software format conversions, but at this writing, there is no automatic method in use to determine the native parameters of an input device. This can lead to connecting an input device with conversions to an output device with conversions, which both wastes CPU time, creates unnecessary duplicate data, and possibly degrades the sound itself.

sound chooses the best format possible for any input. The total number of channels is limited to 8, and the sample rate is limited to 44100 (CD quality sound). ALSA specifies inputs with conversions as having more than 100 channels, which it does by duplicating samples. This is obviously a waste of time and space.

Thus when choosing inputs, look for the devices that say "without any conversions". These devices will have the actual capabilities of the input device instead of "pseudo-parameters", which give the maximum possible parameters that ALSA can fit the input to, with conversions.

One other quirk of ALSA is that it will classify even single channel inputs like a single microphone as two channel devices by sample duplication. ALSA will throw an error for attempts to specify these as single channel devices.

1.12 Network

Network can be used to both connect to remote computers as well as local computers. The local address will be 127.0.0.1, and this will be the address of any server connection. To use network for program to program communication, you can use either TCP/IP or message based communication, and you can use plain text or TLS encrypted communications. A communication to the same machine is considered a reliable message link.

24 Mac OS X specific details

The Mac OS X implementation is experimental at this time.

25 FAQ

Q. Why would I choose Petit-Ami over other major TKs like GTK or QT?

A.

It's really about factors. One internet user said it best: "you're not a major corporation, so you don't matter". Yes, absolutely. If your criteria is backing from a major corporation or armies of programmers, Petit-Ami is not your TK. Past that, the factors are:

1. Petit-Ami is a simpler API with no callbacks, object oriented programming (or "pseudo-object oriented programming") . Its straight C and requires zero (0) extra code over regular C programs.
2. Petit-Ami is compatible with itself. GTK and QT have major incompatible versions. This is a product of their complexity as well as the fact that they can dump compatibility. Why would you put a lot of effort into making an API general and backwards/forwards compatible if you know you can dump it in the trash?
3. Petit-Ami is designed to provide the minimum required features, and not just a "shopping bag" of different features. The result is easier to maintain.
4. Petit-Ami is BSD licensed. It can be used in any project, public or private, non-profit or for profit, open sourced or closed sourced.
5. Petit-Ami has had a stable API for 25 years.

Q. Why did Petit-Ami get translated to C?

A.

Petit-Ami used to be in Pascaline, but most or all of today's operating systems are programmed in C or a C compatible language (like C++ or Objective C). This caused issues with interfacing the two languages and made it difficult to get support with operating systems, since most people don't know or work with Pascaline.

It also makes Petit-Ami more generally applicable, since any C or C compatible system can use it, and there are well known paths to incorporate a C interface into other languages. The original Pascaline interface still exists, it just leaves the lower level details to C code.

Q. Why are there so few widgets? Other TKs have hundreds.

A. Widgets end up providing a large part of the look and feel for an application. Petit-Ami provides the minimum set of simple widgets to get programs running. Its expected that advanced programs will have custom widgets, and this happens even in TKs with hundreds of widgets. In fact, for the example of GTK, a lot of those widgets are part of the desktop environment GTK is used for (GNOME), and may or may not be used by common applications.

A simple way to think of this is that if, say, you are providing 100 different widgets but the underlying operating system only provides 50, you have exceeded the idea of getting the look and feel of the operating system.

The good news is that once the graphics and windowing functions are there, writing a widget is both simple and portable (to other Petit-Ami installations).

26 License