PICA programmer manual

This document gives details about the PICA library and describes its internal code for each platform on which is available.

Index

PICA programmer manual 1

Index 2

1 - Introduction 3

1.1 - Terminology 3

2 - PICA library internal structure 3

2.1 - windows-based operating systems 5

2.2 - Linux 6

3 - Requirements in terms of tools and libraries 7

4 - How to use PICA 7

4.1 - Linux 8

4.2 - Windows operating systems 8

PICA’s primitives 10

4.3 - Memory management primitives 10

4.3.1 In order to write the log file 10

4.3.2 Management packet buffer 11

4.3.3 Pipe management 12

4.4 - Process Management primitives 14

4.4.1 Timer management: 14

4.4.2 Mutex management 22

4.4.3 In order to manage PICA libary 23

4.5 - Communication management primitives 24

4.5.1 In order to get information on available devices 24

4.5.2 Management forwarding information 25

4.5.3 Sending and receiving packets management 25

4.5.4 Routing management 26

4.5.5 Socket 26

4.6 - PICA data structures and costants 26

1 - Introduction

Since PICA is released under the GNU General Public Licence, any programmer can improve PICA. This document was been created to make this task easier: here all the functionality of PICA’s code is explained, and differences between operating systems, and how PICA copes with them, are pointed out.
Moreover, in this document the programmer can find all the data structures and constant definitions (Appendix A).
Some data structures have different definitions for each platform, since Linux and Windows operating systems provide different data types; for example, a file is identified by a integer in Linux and by a HANDLE (a special type) in Windows.
In this document, the data structures are presented for the Windows version. If they are defined in different way in Linux, it is pointed out in the description.
In this manual the programmer can find necessary instruction to install and use (chapter 4 -) and code description of all PICA primitives. These are divided in three logical group: memory management (Section 5.1 -), process management (chapter 5.2 -) and communication management (Section 5.3 -).

1.1 - Terminology

Since the biggest difference between operating systems is between Linux and Windows-related families of kernels, in the document the word “Windows” is to be interpreted as the Windows operating system’s family, including both desktop and compact editions. In the case it is necessary to be more precise we will clearly state the version of Windows being analyzed.

2 - PICA library internal structure

This paragraph shows PICA project internal structure for each platform on which PICA is available.

PICA project is constituted from eleven files of code and each of them includes a header file.
The figure 4.4 shows the general internal structure of PICA project and the relationships between header files and used library.

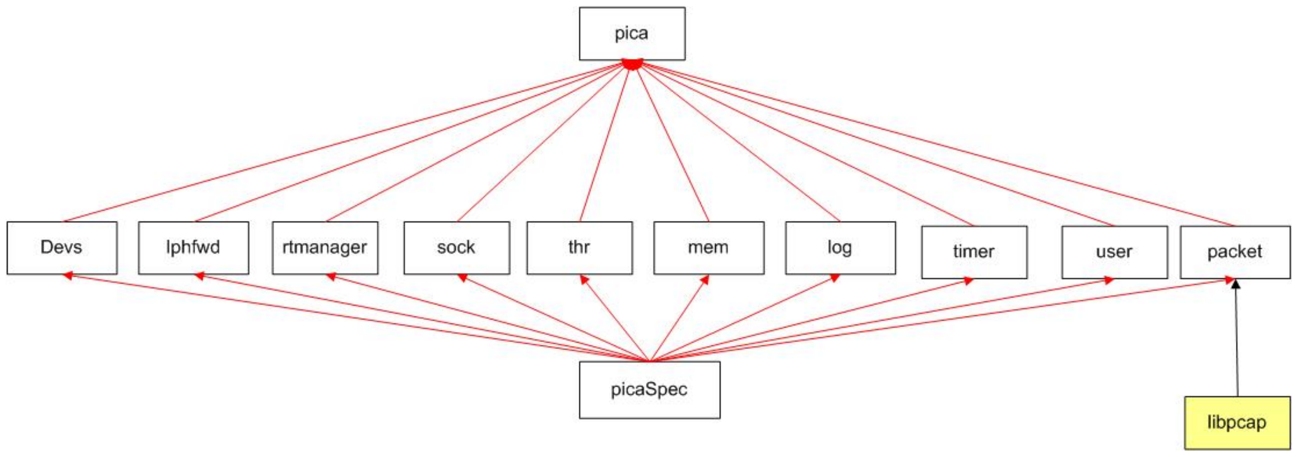
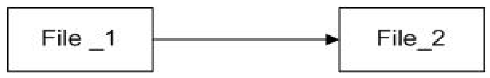


Figure 2-1: PICA internal structure

In this figure a white rectangle is a single PICA header file, while the yellow rectangle is used for Libpcap porting header file.
Arrows are used as follows: for example when we draw



In this case File_2.h includes File_1.h with the directive #include“File_1.h”.

PICA Linux version internal structure is the same shown above, but without references to Libpcap.

Some functions provided by PICA need special libraries in order to interact with operating system and drivers; hence these libraries depend on operating system.

2.1 - windows-based operating systems

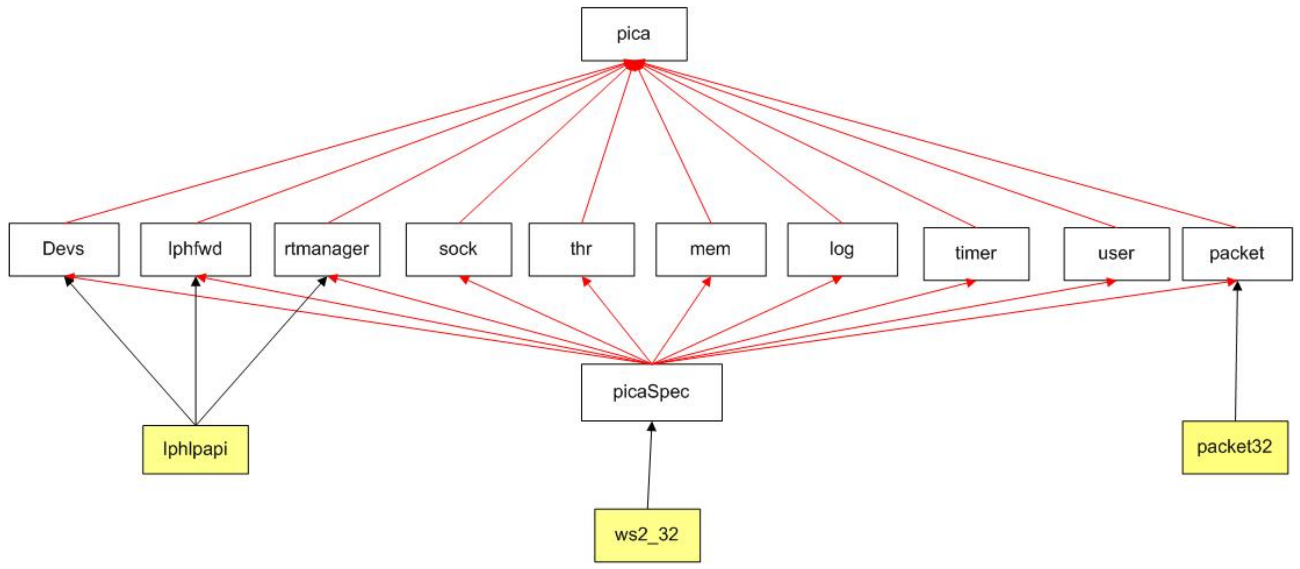


Figure 3.1: PICA internal structure on Windows operating systems

The figure 3.1 shows relevant libraries referred by PICA library for all operating systems based on windows.

The only difference between Windows XP or 2000 and Windows CE is the respectively porting of Libpcap: Winpcap for first type and Packet32 for the second.

Figure 4.5 reports only libraries does not use ordinarily.

Internet Protocol Helper API (Iphlpapi) assists network administration of the local computer by enabling applications to retrieve information about the network configuration of the local computer, and to modify that configuration. IP Helper also provides notification mechanisms to ensure that an application is notified when certain aspects of the local computer network configuration change.

This library is used by PICA in order to get information about available network interfaces, to manage “time to live” (TTL) and forwarding settings, to send and to receive packets.

Ws2_32 Windows socket library (version 2) uses the sockets paradigm that was first popularized by Berkeley Software Distribution (BSD) UNIX.

It was later adapted for Windows in Windows Sockets 1.1, with which Windows Sockets 2 applications are backward compatible.

2.2 - Linux

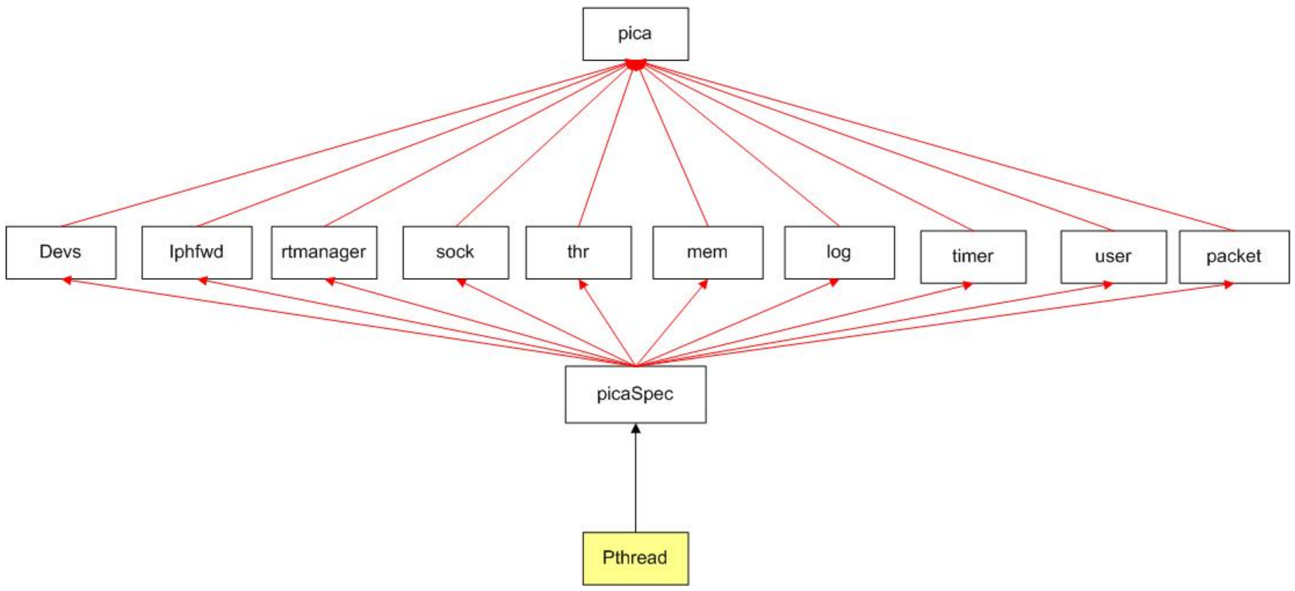


Figure 2-2: PICA internal structure for Linux

PICA Linux version s based on Pthread library.
Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications. In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required. For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995). Implementations which adhere to this standard are referred to as POSIX threads, or Pthreads. Pthreads are defined as a set of C language programming types and procedure calls, implemented with a pthread.h header/include file and a thread library - though the this library may be part of another library, such as libc.
It is fundamental notice that PICA Linux version does not need libpcap. In the first time, PICA was designed based on lipcap in order to use libpcap functionalities to interact with network adapter, but, subsequently, PICA designers opted for using socket feature in place of libpcap.
Sine socket Linux allows a direct interaction with network adapters, PICA designers prefer use it, even if it is a little bit slowly, instead of bind PICA to an other library, augmenting code

size and make it dependent on other code and its new released versions.

3 - Requirements in terms of tools and libraries

In this paragraph we list the necessary tools to manage PICA's code, along with the essential libraries to run PICA.

	Win2000 - XP	**WINCE 3.0**	**WINCE 5.0**	**Linux**
Library	Winpcap winsock2	Packet32 winsock	winsock	-
Tool	Microsoft Visual Studio .Net 2003	ActiveSync 3.5 EmbeddedVisul c++ 3.0	ActiveSync 4.2 Visual Studio .Net 2005	

Obviously, if the programmer does not use PocketPC, solely relying on the simulator provided by development environment, the ActiveSync tool is not necessary. Moreover, ActiveSync 4.2 is back compliant therefore, if the programmer uses it, they can develop code both for Windows CE 3.0 and for Windows CE 5.0.

Tests on PICA were done with the tools' versions shown in the table above; the user can pick a different version, but optimum functionality is not guaranteed.

4 - How to use PICA

The provided PICA package contains these directories:

- Linux
- Windows
- WinCe 3.0
- WinCe 5.0

In each directory there is the source code and the necessary files to work with the tools specified above.

4.1 - Linux

In this directory there are all the PICA files and the Makefile. This file can be used by programmer to compile and install the PICA library in order to use it in other applications.
Respectively, the commands are:

a. make
b. make install

By default, the PICA library is installed in *“/usr/local/lib”* and PICA header files in *“/usr/include/PICA/”*.

Obviously, it is possible change PICA’s directory of installation by changing the paths defined on the Makefile.

All PICA-based applications must include the following code line:

```
#include <PICA/pica.h>
```

And can be compiled with this command:

```
      gcc <your stuff> -lpica
```

4.2 - Windows operating systems

In order to work with the Windows versionof PICA, it is necessary to open each project using the solution file, and be sure that the right references to essential libraries are defined in the project settings.
Moreover, in order to work with Pocket PC (Windows CE), it is necessary to be sure that the development tool downloads all the libraries on the device, and that they are in the same directory.

How to work with PICA on Windows XP:

a. Install Winpcap executing winpicap_setup.exe (the user can download a up-to-date version of this library from: www.winpcap.org/install/default.htm)
b. Open the solution file with appropriate tool and verify that, in the project dependences, are defined ws2_32.lib and IPhlpapi.lib
c. Build pica solution to create the pica dynamic-link library (pica.dll) and static-link library (pica.lib)

How to work with PICA on Windows CE 3.0

a. to install the necessary libraries download them from http://www.winpcap.org/install/default.htmt. The download consists of a zip file that contains a project developed with Microsoft Embedded Visual Studio c++. This solution contains three projects named:
    i. DLL
    ii. Driver
    iii. SampleApply.

While the first two projects consist of code to allow direct interaction with network interface, the third is a small application that demonstrates the functionality of the first two libraries.
In order to obtain the packet32 library it is sufficient to build the DLL project and download it to the PocketPc. Since it is dependent on the Driver project, the same action must be performed on this library also. Notice that output files of packet32 project are packet32.dll for DLL and pktdrv.dll.
Since the winpcap's developer does not guarantee its correct functioning on all devices, it is advisable execute SampleApply to check if the driver is suitable to be used on the PocketPC.

c. In PICA's project directory there is an MSinclude directory containing all header files for the packet32 library.
d. Verify that in PICA dependences are defined: IPhlpapi.lib, winsock.lib, packet32.lib and pktdrv.lib

e. Build the pica solution to create PICA’s dynamic-link library (pica.dll) and static-link library (pica.lib)

How to work with PICA on Windows CE 5.0
The winpcap port for Windows CE 5.0 is not yet available. Hence, only these steps must be performed:

a. Open the solution file with the appropriate tool and verify that, in the project dependencies, winsock.lib and IPhlpapi.lib are defined.
b. Build the pica solution to create pica’s dynamic-link library (pica.dll) and static-link library (pica.lib)

5 - PICA’s primitives

This paragraph shows and describes PICA’s primitives divided in three logical groups:

- memory management primitives
- process management primitives
- communication management primitives

On each of them PICA’s primitives are separated by their functionality.

5.1 - Memory management primitives

5.1.1 In order to write the log file

In order to unify the file descriptor, PICA uses “FDesc”, that in Windows is a HANDLE type while in Linux it corresponds to the “int” type.

These functions are wrappers those of the operating system.
In Windows file permission are generic, while in Linux these are “-rw-r- -r- -”.

In windows it is important to notice that, in order to implement file opening functionalities in append mode it is essential to shift the pointer file to the end using the SetFilePointer function provided by Windows, before starting to write.

```
int PICAopenFile(FDesc * file, char * name, int read_write, int flags)
int PICAwriteToFile(FDesc file, void * data, unsigned int datasize);

int PICAreadFile(FDesc file, void * buf, int buffersize, int * datasize);

int PICAcloseFile(FDesc file);
```

5.1.2 Management packet buffer

Relatively to memory management, PICA's architecture is based on offering the possibility to easily manage a data structure to handle a set of queues.

```
int PICAinitBuffer(PICAbuffer ** ibuf, int num_queues);
```

For Windows operating systems it is important notice that, in order to create a mutex that can be inherited by threads, the "CreateMutex" function must be called with the first parameter set to a pointer to a "SECURITY_ATTRIBUTES" structure where field "bInheritHandle" is set to true.
On Windows XP this function creates a PICAbuffer with a number of queues equal to *num_queues*; in order to allocate the queues and an array of mutexs, it uses the GlobalAlloc primitive in place of the classic *malloc* function since Windows XP can generate errors on memory management when an application tries to access a memory location allocated by a library.
Obviously, it is necessary use GlobalFree primitives to free previously allocated memory.

Since in Linux no such problems with memory allocation where detected, PICA uses malloc primitives to allocate memory.

```
int PICAaddToBuffer(PICAbuffer * buf, int queue_id, void * data, int data_size);
```

```
int PICAgetFromBuffer(PICAbuffer * buf, int queue_id, int num_packets, PICApacket ** packets, int * avail_packets);
```

```
int PICAkillBuffer(PICAbuffer * buf);
```

This function un-allocates the memory allocated with PICAinitBuffer.

First of all, it blocks all mutexes in order to prevent the concurrent access problem.

Since mutexs are contained in the structure, the function does a copy of the mutex array and blocks copy, so that it can free the whole memory allocated for the buffer.

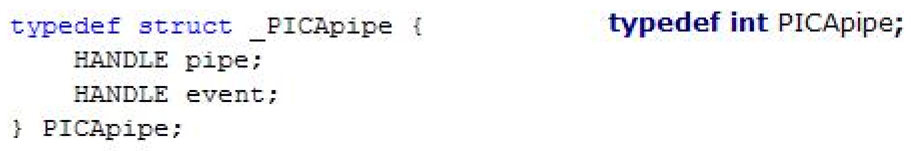
Afterwards, for each queue, the function un-allocates each packet data, and then the whole packet.

Finally, after freeing the whole buffer data structure, the functions frees the mutex array copy.

5.1.3 Pipe management

Windows, unlike Linux, leaves pipe event management to the programmer. When a thread writes on a pipe, it signals this action to the thread waiting for this event; in Linux, operating system delivers signal to resource waiting for; that is, this mechanism is transparent to programmer. Instead in Windows, programmer must set an event for in and out pipe in order to allow waiting thread on in pipe is advised when something is written on out pipe.For this reason, field “event” of “PICApipe” was added.

Hence, PICA provides a pipe data structure and primitives to manage it in order to cope with this difference.
In fact in Windows PICApipe is declared as:



```
typedef struct _PICApipe {
    HANDLE pipe;
    HANDLE event;
} PICApipe;
```

```
typedef int PICApipe;
```

Figure 5-1: PICApipe structure definitions (left: Windows, right: Linux)

```
int PICAmakePipe(PICApipe * in, PICApipe * out);
```

Both in Windows and in Linux functions the PICA pipe function called the respective on made available by the operating systems. It is important to notice that in Windows the system function “createPipe” is called using the “SECURITY_ATTRIBUTES” structure as the last parameter, which must have the “bInheritHandle” field set true; this way theads can inherit it. Moreover, this function creates an event shared by the in and out pipe.

```
int PICAsendToPipe(PICApipe out, void * data, int size, int * written);
```

In order to write data on a pipe both Linux and Windows operating systems use the same functions used for writing on a file.
In Windows, in order to signal writes and reads on pipe, it is necessary use the OVERLAPPED structure as the last parameter of the Windows write function.
By setting the OVERLAPPED structure’s field named “hEvent” with the event of pipeOut, the thread waiting to read on the pipe is informed about the write.

```
int PICAgetFromPipe(PICApipe in, void * buf, int bufsize, int * datasize);
```

In order to write data on a pipe, both Linux and Windows operating systems use the same functions used to read a file.

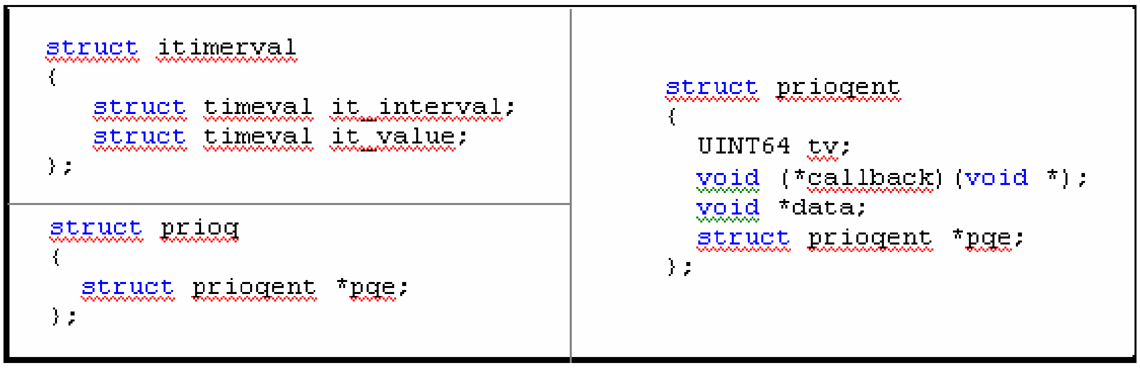
```
int PICAclosePipe(PICApipe pipe);
```

This function frees pipe resources.

5.2 - Process Management primitives

5.2.1 Timer management:

PICA provides multiple timers by means of a priority queue. In this queue all events make use of the same timer in a way that only the first to-happen event affected will be used to set the value of the timer. To perform this task PICA employs the following data structures:



```
struct itimerval
{
    struct timeval it_interval;
    struct timeval it_value;
};

struct prioq
{
   struct prioqent *pqe;
};

struct prioqent
{
   UINT64 tv;
   void (*callback)(void *);
   void *data;
   struct prioqent *pqe;
};
```

Figure 5-2: data structures definition for timer

By starting the up timer, the global variable “pq” with type prioq is allocated.

Each PICAtimer call with parameter “action” equal to “SET” adds a new element of type “priorquent” to the timer queue, with fields set in this way:

- tv: will be set according to the value of parameter “time”
- callback: will point to the function to call when tv time is expired
- data: will point to data to used as a parameter for the function
- pque: will point to the next element.

The elements in the queue are in increasing order: the first one has the lowest timeout value.

Therefore, element are inserted in queue according to their timeout value, maintaining the required ordering.

Moreover, it is important to notice that only the Windows version of PICA defines the itimerval structure, because it is already defined in the Linux header file “time.h”.
All operations on the timer queue are performed by using a mutex avoiding racing conditions. Therefore, even in multi-threaded environments, it is possible use the same timer queue without synchronization problems.

```
UINT64 PICAgetCurrTime(void);
```

As expected, Linux and Windows get current time information using different procedures.
Windows uses GetLocalTime function that returns a structure containing all the information about current time. Windows documentation suggests converting this structure into another one, and this to an unsigned integer of 64 bits in order to perform arithmetic operations on its value.
Therefore, Windows code is not a sample call of Windows system functions, but contains all the necessary instructions to perform this conversion.
In Linux, PICA’s code calls the appropiate system’s function and transforms its result into an unsigned long integer in order to unify types with Windows.

```
int PICAtimer(int action, UINT64 * time, void * function, void * data);
```

This function is implemented in the same way both in Windows and Linux, but system calls are different, obviously. Here we describe the code for the Windows version, and if here is some differences towards the Linux version, they are pointed out.
Behaviour of this function depends on the value of parameter “action”, like it is explained in PICA’s user manual, chapter x.x.

If “action” ‘s value is “STARTUP”, the function initializes the global variable *pq* by “malloc” invocation and by setting its value to NULL. In addition, the Linux version of PICA starts the thread that manages timer.

If “action”’s value is “T_SET”, the function calls the “pq_insert” function. ”pq_insert” handles the insertion of a *prioqent* element in the list and also calls “pq_updatetimer”. This function creates the thread (identified by “timerHandle”) that manages timer. Afterwards, it sets the timer value depending on these situations:

- if the queue does is empty, the timer is set to zero;
- if the queue contains one or more elements:
  - if the first event timeout is lower than the current time then the timer’s value is of one microsecond. (this is the case that the event has just happened)
  - otherwise, the timeout value is the differences between its timeout and the current time.

The function, named “TimerThrFunc”, that is executed by the thread is quite similar for both Linux and Windows environments.
First of all, the thread creates the timer. It then enters an infinite loop where it performs the following tasks:

a. check if the timeout value for the first element is close to the current time. It is obvious that it is impossible to activate an event at its exact timeout value since it is expressed with microsecond precision. Therefore, PICA actives an event if its timeout is in the range between its timeout plus or less 100 microseconds.
b. If the previous checking returns true, then the thread executes the event’s function and afterwards deletes it from queue.
c. Re-start the loop

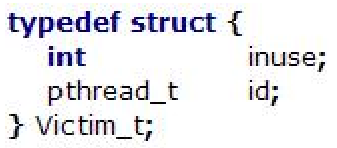
If “action”’s value is “T_STOP”, the function searches the element identified by its other parameter; this task is performed by “pq_getfromqueue” and “pqdeleteen”. The first element found that matches the criteria is deleted. Finally, PICAtimer calls the “updatetimer” function.

If the “action”’s value is “T_KILL”, the function frees the memory allocated for all the variables used by the timer by calling *pq_cleanup*.

In Linux, this function calls thread in a different moment: while in Windows it is created on the first call of PICAtimer with “T_SET”, in Linux it is created in a PICAtimer call with T_START; this different does not change this function’s behavior, but its due to differences in terms of thread and resource management for the different operating systems.
Thread management
The fork() call is of common use in Unix environments to manage processes. The Windows OS, though, does not offer this function. PICA adopts a combination of the threads approximation along with the semaphore and mutex abstractions as an alternative to processes without generating too much extra code. Although the Posix standard doesn’t allow thread suspension and resuming, the PICA library allows to use such functions in the Linux operating system by means of the SIGUSR1 and SIGUSR2 signals.

PICA’s Windows functions focusing on threads merely wrap the operating system’s primitives, but it is important to notice that the Windows functions called in PICASuspendThread and PICAResumeThread are primarily designed for use by debuggers, and they are not intended to be used for thread synchronization.
PICA’s Linux functions are quite more complicated than than those for Windows since Linux does not provide thread suspend and resume functionality.
Before analyzing each PICA function code concerned thread, here we report the fundamental idea on which these functions are based.

Since Linux does not admit thread suspending and resuming, it does not provide a way to store information about which threads are suspended in order to call the resum function on them alone.
 Therefore, PICA faces up this problem using a global dynamic array named ”array” of type “Victim_t”, a new type introduced by PICA.



```
typedef struct {
    int          inuse;
    pthread_t    id;
} Victim_t;
```

Figure 5-3: Victim_t structure definition

The first field is used to know if array element are in use or not. By default, PICA defines “array” with 10 elements but, if necessary, this array can easily grow in size.Obviously, The “id” field identifies the thread.
Moreover, PICA uses global semaphore variables in order to allow the thread to communicates to the main process its entry in the suspend state.

```
int PICAstartThread(THRID * thr, void * func, void * arg);
```

This function creates a detached thread, that is its thread ID and other resources can be reused as soon as the thread terminates.

```
int PICAsuspendThread(THRID thr);
```

First of all, PICAsuspendThread initializes all the necessary data structures and variables for thread suspension by invoking “pthread_once”. This function calls “suspend_init_routine” that allocates an “array” variable, initializes the semaphore and installs the signal handlers for suspending and resuming through an “sigaction” call.
It important to notice that “suspend_init_routine” is called only once by definition of the “pthread_once” function.

Afterwards, PICAsuspendThread checks if the thread identified by “thr” is already suspended by searching “thr” in “array”.
If “thr” is not already suspended, this function sends SIGUSR1 to the thread to suspend it and, then, enter an wait state on semaphore. The main process stays

in the wait state until the thread enters in suspend state. Hence, the thread executes “suspend_signal_handler”, the routine assigned to the SIGUSR1 signal in the previous initialization phase.

“suspend_signal_handler” suspends all signals for the thread except for SIGUSR2, and communicates its entering in suspend mode to the main process by calling“sem_post” on the global semaphore.

When the main process receives this communication, it exits from the wait state and continue its execution.

int PICAresumeThread(THRID thr);

First of all, this function calls “pthread_once”. (for more details see the previous function).

Subsequently, it checks if the thread identified by “thr” is in the suspend state. In the negative case, it returns with error; otherwise, it sends the SIGUSR2 signal to the thread through the “pthread_kill” function invocation and then returns successfully. When the thread receives that signal, it executes “resume_signal_handler”. This handler is a sample return but it is necessary in order to force “sigsusped” function, called by the thread when it is entering the suspend state, to return.

int PICAkillThread(THRID thr);

In both Windows and Linux this function is a wrapper of the one made available by the operating system.

int PICAselect(int time, PICAdescList * dl, PICAselResult * res);

In Linux any resource is identified by an integer descriptor. Therefore, since the select function provided by Linux works with resource descriptors, it can be use for all sorts of resources.

On the contrary, for Windows operating systems, descriptors are generally represented by a specific data type called HANDLE, while integers are only used for sockets. The PICA library obviates this problem by emulating Linux’s behaviour

The fundamental idea of the algorithm used for the Windows version of PICA is to separate socket descriptors from others using a

thread that calls the Windows select function on those sockets inserted in the list; simultaneously, the main application perform a select on the other descriptors.
The thread can communicate its results to the main application through a pipe.
The code in detail:
The first while-loop divides descriptors contained in *dl* in two lists: in the first one, named “sock”, we have all socket descriptors, while in the second one, called “handles”, there are all remaining descriptors whose type is “PICA_PIPE_TYPE” and “PICA_OTHER_TYPE”. After exiting this loop, the function creates a new event to be added to the second list. This event corresponds to a thread’s write on the pipe.
Before creating the thread, Windows select function is called with timeout 0 in order to check if any of the sockets is ready. In such case the function scans the “sock” list until it finds the first signalled socket, and then sets variable “res” (whose type is PICAselResult) with the values of the correspondent fields of the signalled socket descriptor, and finally returns.
If there are no ready sockets, the function creates pipe and thread. This executes the code of the “sock_select” function. This is a simple Windows select function call with the timeout value set as parameter “time” of PICAselect.
Hence, there is a parallel execution of the thread and the main application.
Notice that, while the thread waits on sockets, the main application waits on other types of descriptors. In order to attempt to do this task, the main application uses the “WaitForMultipleObjects” function. Since this function required an array of objects on which to wait upon, the second while-loop copies the “handle” list in handleArr array.

If PICAselect is called with “PICA_WAIT_FOREVER” as the actual parameter of “time”, the “WaitForMultipleObjects” is invoked with an infinite timeout. Otherwise, it uses the value of parameter “time”.

If the call to *WaitForMultipleObjects* has success, then it returns an integer included between WAIT_OBJECT_O and WAIT_OBJECT_O plus handlerArr elements number.

If the value is equal to WAIT_OBJECT_O it means that the pipe used for intercommunication between the thread and the main application sends a signal, because its descriptor was been saved as the first element of handleArr array .
In this case, PICAselect can find on the pipe the index of the socket that has been signalled, and then set the “res” variable field with the signalled socket value.

If WaitForMultipleObjects has success, and the returned valued (said i) is bigger than WAIT_OBJECT_O, the fields of the “res” variable will be set with the descriptor values at the “i^{th}” position in the handle list.

In Linux, the PICAselect function is a simple call to the select function provided by o.s.

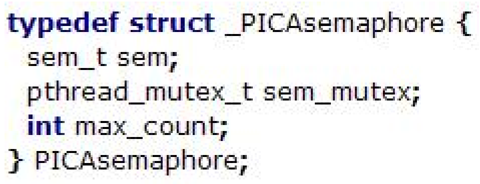
Before invoking the select function, PICAselect cleans the set of signalled descriptors by using the FD_ZERO function, and then creates a new set with all descriptors contained in “dl” list by invoking FD_SET.

The time rule is the same on as for Windows: if PICAselect is called with “PICA_WAIT_FOREVER” then the select function is invoked with the last parameter set to null in order to return only when a descriptor is signalled.
If time is other number, this value represents the timeout for the select function.

```
int PICAaddDesc(PICAdescList ** dl, int type, int mode, void * desc);
```

5.2.2 Mutex management

Windows and Linux manage mutexes and semaphores in different way. Besides having different types of mutex and semaphore descriptors, Linux does not allow setting the maximum value for a semaphore, which is a feature available in Windows. PICA provides this functionality.
In order to accomplish this, it was necessary create a new data structure for semaphores in the Linux version:



```
typedef struct _PICAsemaphore {
  sem_t sem;
  pthread_mutex_t sem_mutex;
  int max_count;
} PICAsemaphore;
```

Figure 5-4: PICAsemaphore structure definition for Linux

In this structure, the first field is the real Unix semaphore, the second is a mutex used to allow secure access to “max_count” field that stores the maximum semaphore value.
The PICAcreateSemaphore consists of a call to the “sem_init” function (functionality provided by Linux in order to initialize semaphores) and setting all field of “p_sem”, its first parameter.

```
int PICAcreateMutex(PICAmutex * mut);
```

```
int PICAcreateSemaphore(PICAsemaphore * p_sem, int initial_count, int max_count);
```

This function consists of two system calls to initialize semaphores and mutexes (sem_init and pthread_mutex_init, respectively) and in setting the “p_sem” variable according to the values of its parameters.

```
int PICAmutexAction(int action, PICAmutex * mut);
int PICAsemaphoreAction(int action, PICAsemaphore * p_sem, int count);
```

The only thing to notice in this function's code is about its action when the action parameter takes the value SEMAPHORE_ACQUIRE.
PICAsemaphoreAction calls "count" times operating systems function that allows increment semaphore value, because neither Linux neither Windows offer functionality to increment that count.

```
int PICAdestroyMutex(PICAmutex * mut);
```

```
int PICAdestroySemaphore(PICAsemaphore * p_sem);
```

In order to get user information

```
int PICAisAdministrator(int * true_false);
```

5.2.3 In order to manage PICA libary

```
int PICAstartup(int flags);
```

Both Linux and Windows initilize global variables by using the respective system functions; but it necessary to point out that Windows needs to call The WSAStartup function. This initiates use of the Winsock DLL by a process.

```
int PICAcleanup(void);
```

The following functions are used to manage errors in the PICA library.
The first one is used in PICA's code each time an error occurs. When this happens, the message and error codes are saved in the global variables defined in PICAspec.c file, err_buf and err_code, respectively.
Both functions use a mutex to access them, thereby avoiding concurrent access problems.

```
void P_ERROR(char * message, int code);
int PICAgetLastError(char * err, int * code);
```

5.3 - Communication management primitives

In Linux, all PICA primitives related to network functionality are implemented by invoking the ”ioctl” function, which manipulates the underlying device parameters of special files. In particular, many operating characteristics of special character files (e.g. terminals) may be controlled through ioctl() requests.
The first parameter the ioctl function is a descriptor file, while the second represents a command that selects the control function to be performed and shall depend on the STREAMS device being addressed.
Therefore, in Linux, these PICA functions are implemented by invoking the ioctl function on a socket, and with different commands in order to get or set the necessary information.

5.3.1 In order to get information on available devices

In Windows these two functions are implemented by calling GetAdaptersWindows, provided by the IPhlpapi library. The Windows primitive is invoked twice. The first call is used to find the number of available devices, while the second is used to obtain information about the device.
In PICAgetAvailableDevices the information concerns devices names, while in PICAgetDeviceAttrs it consists of IP and MAC addresses.

In Linux….

```
int PICAgetAvailableDevices(DEVLIST * devs);

int PICAgetDeviceAttrs(char * dev, DevAttrs * attrs);
```

5.3.2 Management forwarding information

These functions provide information about forwarding and TTL, also allowing to change their states.
In Windows, in order to attempt this task they use “GetIpStatistics”, a function provided by the Iphlpapi library.
Forwarding and TTL information are stored in structure “MIB_IPSTAT”; therefore, in order to change the forwarding status or the TTL value, it is necessary to change forwarding and/or TTL values and then call “SetIpStatistics” with this structure altered according to the new settings.

Since in Linux all settings are stored on file, this function reads and writes the appropriate files to get and set their values, respectively. Since these files are system files, pica must be executed with root priviledges.

```
int PICAisForwarding(int * true_false);
```

```
int PICAsetForwarding(int on_off);
```

```
int PICAdefaultTTL(int set_get, int * ttl);
```

5.3.3 Sending and receiving packets management

In Windows these functions use primitives provided by winpcap.
Hence, to understand the code for Windows please consult winpcap’s documentation. Linux code relies on socket functionality for this task.However it is necessary explicate wince 3.0 code…..

```
int PICAopenDevice(char * device, PICA_IO_DEVICE * iodev);
```

LINUX: open a socket and bind

```
int PICAframe(int mode, PICA_IO_DEVICE iodev, void * packet, int packetsize, int
* read);
```

```
int PICAcloseDevice(PICA_IO_DEVICE iodev);
```

```
int PICAcreatePacket(char *addr, unsigned char *data, int datasize, unsigned char * packet, int * packetsize);
```

5.3.4 Routing management

WINDOWS: these functions are wrapper operating system one.
LINUX: uses a socket on AF_INET domani and a ioctl call with command SIOCADDRT

```
int PICAaddRoute(UINT32 dest, UINT32 mask, UINT32 gateway,int metric, char * device);
```

```
int PICAdelRoute(UINT32 dest, UINT32 mask, UINT32 gateway, char * device);
```

```
int PICAgetRoutingTable(RTInfo * rti);
```

5.3.5 Socket

These functions are only wrapper operating system ones.
The different between Windows and Linux are the command to make socket no blocking.

```
int PICAcreateSocket(PICAsocket * sd, int domain, int type, int protocol, int block);
int PICAcloseSocket(PICAsocket sd);
```

5.4 - PICA data structures and costants

This paragraph resume all data types provided by PICA.
See file PICA_data_structure.doc

PICA defines following constants:

DEVS

```
#define MAXDEVS 32
#define MAXDEVSIZE 128
```

IPFWD

```
#define FWD_ON 1
#define FWD_OFF 0
#define TTL_SET 1
#define TTL_GET 0
```

LOG

```
#define READF 0
#define WRITEF 1
#define READF_WRITEF 2
#define CREATE_CLEAN 0
#define APPEND 1
```

MEM

PACKET

```
#define PICA_SEND 0
#define PICA_RECEIVE 1
#define PROMISCUOUS 0
#define ALL_LOCAL 1
```

PICASPEC

```
#define PICA_WINDOWS_NT
#define PICA_VERSION 0x010000  //version 1.0.0
#define PICA_ERR_BUF_SIZE 100
```

RTMANAGER

```
#define BUFSIZE 3000
```

SOCK

```
#define NO_BLOCK 0
```

THR

```
#define PICA_SOCKET_TYPE 0
#define PICA_PIPE_TYPE 1
#define PICA_OTHER_TYPE 2
#define PICA_TIMEOUT_TYPE 255

#define PICA_WAIT_FOREVER -1

#define MUTEX_ACQUIRE 0
#define MUTEX_RELEASE 1
#define MUTEX_ACQ_NO_BLOCK 2

#define SEMAPHORE_ACQUIRE 0
#define SEMAPHORE_RELEASE 1
#define SEMAPHORE_ACQ_NO_BLOCK 2
```

TIMER

```
#define TIME_DIV 100

Linux : #define TIME_DIV 10


#define T_SET 0
#define T_STOP 1
#define T_STARTUP 2
#define T_KILL 3
```

USER

```
#define IS_ADM 1
#define IS_NOT_ADM 0
```