In this document, basic information for the usage of pcynlitx software development platform has been given

The author: ERKAM MURAT BOZKURT, M.Sc in Control Systems Engineering

e-mail: pcynlitx.help@gmail.com

INTRODUCTION TO PROGRAMMING WITH PCYNLITX

1 A BASIC INTRODUCTION TO THE PCYNLITX PLATFORM

Pcynlitx is an outcome of the scientific research study that is carried out by Erkam Murat Bozkurt. Basically, pcynlitx acts as a separate intelligent actor simplifying the multi-threaded software development process. The main purpose of this research study is to develop an easy and efficient way in order to decrease the complexity that are faced on the programming with low level languages such as C++. Basically, pcynlitx is just a particular application of this new paradigm and it is an open source, free to use IDE which produces a class library for C++ multi-thread programming applications. The outcome of the pcynlitx acts as an autonomous management system for the thread synchronization tasks. Although the idea behind the pcynlitx platform is complex, the usage of the pcynlitx is very easy.

2 THE BIGGEST PROBLEM ON MULTITHREADING

In multithreading, the biggest problem is non-deterministic scheduling of the threads. The execution order and the priority of the threads reflect the term of scheduling. More specifically, it is a decision about which thread will run and which thread will be suspended in a particular time interval. In practice, the operating system schedules the threads. On the contrary, on the applications in which pcynlitx platform are used, the execution order of the any code section can be determined.

2.1 What is meta-programming

Meta-programs are special type of programs that writes codes. In other words, they are the softwares that write codes. Actually, pcynlitx is a collection of several meta-program and a graphical user interface (GUI) that controls the execution of these meta-programs. In pcynlitx platform, the programmers must develop the programs step by step and in each step, the pcynlitx gives some reports to the user. More specifically, in pcynlitx, the programs are developed by means of an interaction between the programmer and the pcynlitx. From a different point of view, we can easily say that the pcynlitx assists the programmer in the multi-thread application development.

2.2 Introduction to Descriptor File

Pcynlitx is not a multi-threading library. Instead, it is a multi-threading library generator and in each library construction process, it produces a new "project-specific multi-threading library" relaying on the programmer needs. The term "application specific multi-threading library" refers such a library that has some project-specific properties. In other words, on the library, there are some special tools which are generated for a certain class or application. However, in order to achieve this, pcynlitx needs to improve its global knowledge about the software to be developed. Therefore, before the library construction process, pcynlitx collects information about the context of the program from the programmer by means of a descriptor file. However, this communication brings some extra costs for the programmer. In order to differentiate each information from the others, some special syntax rules has to be followed by the programmer when information entered to the descriptor file. A sample description that is used on the descriptor files has been shown in below.

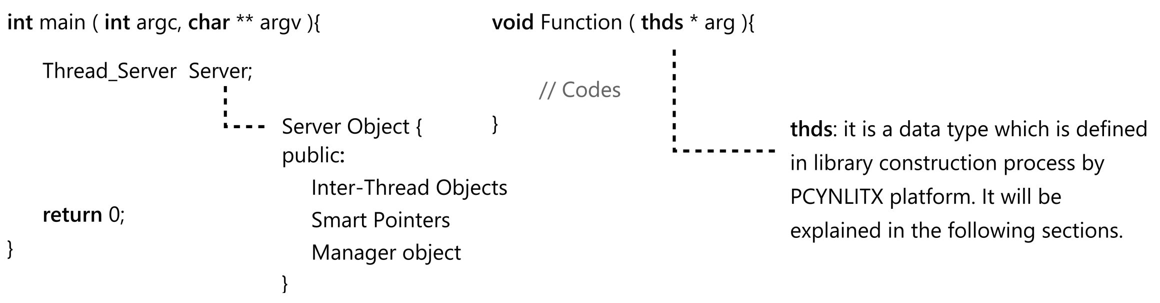
```
Description [ Thread_Number ] {

      4
}
```

Figure 1: A typical pcynlitx description

For descriptor file, the word description is a key word and it indicates that the following code lines include descriptions about multi-threaded program. Fortunately, the pcynlitx users does not have to learn the coding syntax of the descriptor file. In contrast, the users enter their preferences by means of the graphical user interface and the descriptor file is coded automatically. Then, before the library construction process, the users can easily print their own preferences to the screen. On the other hand, if the descriptor file will be written by the programmer, the name of the descriptor file must be determined as "Project_Descriptor_File" in the file system. Different file names can not be recognized as descriptor file by the pcynlitx. In practice, descriptor files act as an interface between the programmer and pcynlitx. From a different perspective, descriptor file can be considered as a contract which is signed between the meta-program and the programmer before the library construction process.

3 THE CLIENT-SERVER TAXONOMY IN PCYNLITX

In pcynlitx, the operations between the threads and the main thread are performed based on a client-server taxonomy. In this taxonomy, it is assumed that the main thread is the server of the variables to be shared between the threads and the threads to be created are the clients of the main thread. In fact, on the pcynlitx applications, only the addresses of some pre-defined objects and/or smart pointers can be shared between the threads. The reason of this restriction is the optimum memory usage. The parameter pass operations are performed by a special object in pcynlitx applications. In fact, this object is a composition of the objects in which their addresses are shared between the threads. In pcynlitx applications, this object is named as the server object and the class which defines the type of the server object is named as server class as well. Strictly speaking, the server object passes the addresses of its own members to the threads automatically and it must be instantiated on the main function when it is used. In pcynlitx applications, the therm inter-thread object is an abbreviation that indicates the objects which are shared between the threads. In pcynlitx, if an object will be shared between the threads, the necessary informations about the class defining the type of this object must be declared on the descriptor file. Default name of the server class is "Thread_Server" and it is determined by the pcynlitx. However, the user of the pcynlitx can also set the name of the server class. The basic structure of the pcynlitx applications is given in below.



```
int main ( int argc, char ** argv ){

    Thread_Server  Server;

    return 0;
}

Server Object {
public:
    Inter-Thread Objects
    Smart Pointers
    Manager object
}

void Function ( thds * arg ){

    // Codes

}
```

thds: it is a data type which is defined in library construction process by PCYNLITX platform. It will be explained in the following sections.

Figure 2: The structure of the pcynlitx applications

In the figure which is given in above, it is seen that there is another server object's standard member which named as "Manager Object". The "Thread_Manager" class is the type of the objects which are used in order to synchronize the threads and an instance of the Thread_Manager class (*Manager object*) is defined as a public member of the server class on the each library construction process automatically. On the following of the tutorial, the manager object will be explained. For now, we must concentrate the server object. It is obvious that an instance of each member of the server object is produced automatically when an instance of the server class is constructed on the main function. Then, each public member of the server object will be defined on the main function scope. It has been already mentioned that only the addresses of the inter-thread objects are shared between the threads. However, in the library construction process, some special classes in which their instances are used as a connection point to the inter-thread objects are defined automatically by the pcynlitx platform. In pcynlitx applications, these classes are named as client classes and their instances are named as client objects. The client classes are explained briefly in the following of this document.

4 HOW THE THREADS CAN BE ACTIVATED IN PCYNLITX

In pcynlitx, many operation are performed by server object and thread creation is performed by the server object as well. The server class has a member function that creates the threads and this function is named as “Activate( )”. With the help of this member function, the threads can be created with their unique numbers that are given by the programmer. A sample thread creation is shown in the Figure that is depicted in below. In pcynlitx applications, the threads are controlled by means of their unique numbers which are assigned by the programmer to the threads. Hence, in the figure, the first argument is the number of the thread to be created and the second argument is the name of the thread function to be executed. The type of the first argument is integer and the type of the second argument is a function pointer that points the thread function routine. Simply,it is the name of the function routine which will be executed by the thread to be created.

5 HOW THE PARAMETERS ARE PASSED TO THE THREADS

Pcynlitx defines a data structure including the pointer members that point the inter-thread objects and smart pointers (*the objects which are shared between the threads*). This data structure is named as inter-thread data structure (*itds*). The

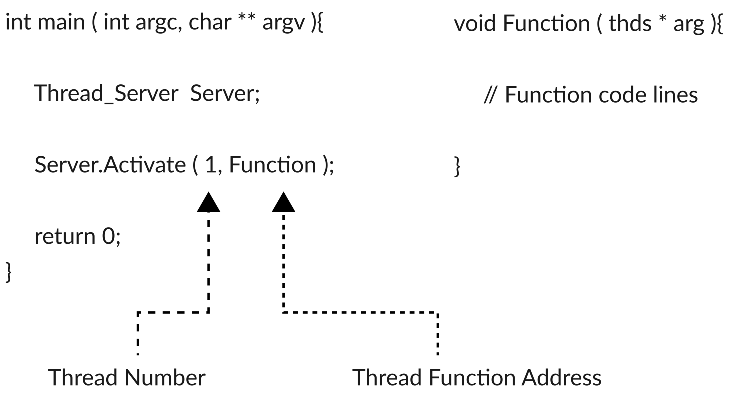

int main (int argc, char ** argv){
Thread_Server Server;
Server.Activate (1, Function);
return 0;
}
void Function (thds * arg){
// Function code lines
}
Thread Number
Thread Function Address

Figure 3: Activation of the threads in Pcynlitx applications

server class includes an instance of that structure which its members point the inter-thread objects. It has been already indicated that the inter-thread objects are also the members of the server class. Therefore, with the help of the inter-thread data structure, the server object can pass the addresses of its own members to the threads. To be able to pass the unique numbers which are given by the user to the threads, in the the library construction process of the pcynlitx applications, a second data structure which is named as the thread specific data structure (*thds*) is defined automatically. Thread specific data structure includes both thread number and a pointer that holds the address of the inter-thread data structure. This mechanism is shown in below figure.

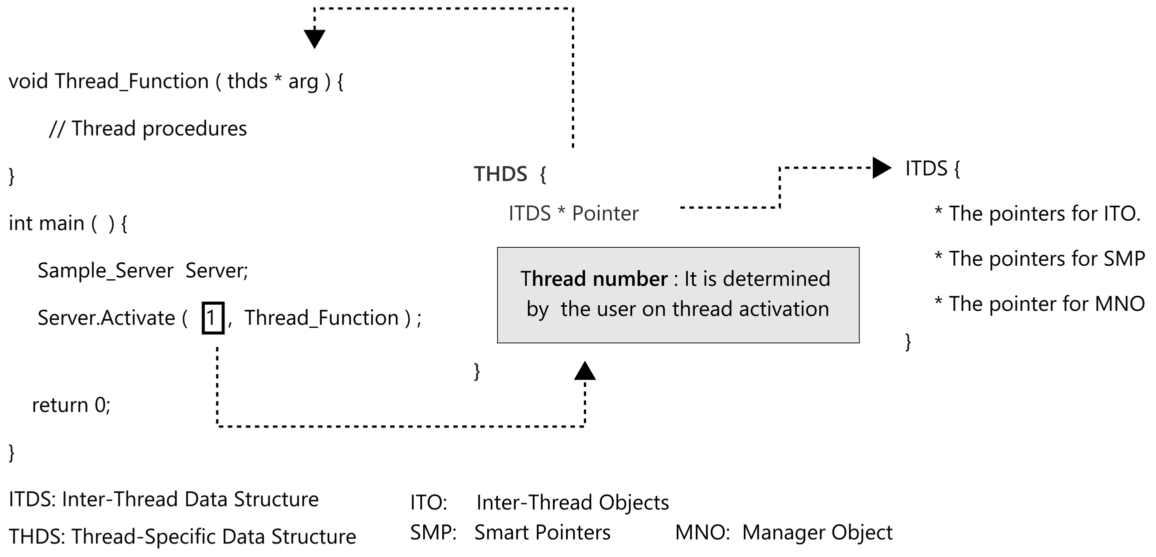

void Thread_Function (thds * arg) {
// Thread procedures
}
int main ( ) {
Sample_Server Server;
Server.Activate (1 , Thread_Function) ;
return 0;
}
THDS {
ITDS * Pointer
Thread number : It is determined by the user on thread activation
}
ITDS {
* The pointers for ITO.
* The pointers for SMP
* The pointer for MNO
}
ITDS: Inter-Thread Data Structure
THDS: Thread-Specific Data Structure
ITO: Inter-Thread Objects
SMP: Smart Pointers
MNO: Manager Object

Figure 4: The parameter pass mechanism of the pcynlitx applications

If the address of the inter-thread data structure is passes to the threads, then, each inter-thread object will be reachable on the threads. But, it is still unclear that how the public member functions of these inter-thread objects can used on the address spaces of the threads. In the library construction process of the pcynlitx platform, a helper class which can be used as an indirection mechanism is produced automatically for each inter-thread class. In the previos sections, these classes are indicated named as the client classes. These special classes have a copy of each public member function of their corresponding inter-thread class and the indirection are performed automatically on the inside of these member functions. By this way, the programmer can call the public member functions of the inter-thread objects from the address spaces of the threads. In pcynlitx, the relation between an inter-thread object and its helper object can be considered as a “client-server” relation. Because, helper objects are just connection points to the inter-thread classes. Whenever a helper class’s member function is called, actually its parameters are passed to its corresponding inter-thread class. The client classes is named with the combination of the name of its inter-thread classes and "_Client" subfix.

6 HOW THE CLIENT CLASSES ARE USED ON THE PCYNLITX APPLICATIONS

The constructors of the client classes take the only argument of thread functions which is in fact the address of the thread spesific data structure. For example, let we assume that the name of the inter-thread class is "Mean_Value" and the name of its client is "Mean_Value_Client". The instance of the Mean_Value_Client can be constructed as in below.

```
void Function( thds * arg ) {

        Mean_Value_Client Sample( arg );

        std::cout << "The mean value:" << Sample.getMeanValue( ) ;
}
```

Figure 5: The usage of the client classes in pcynlitx applications

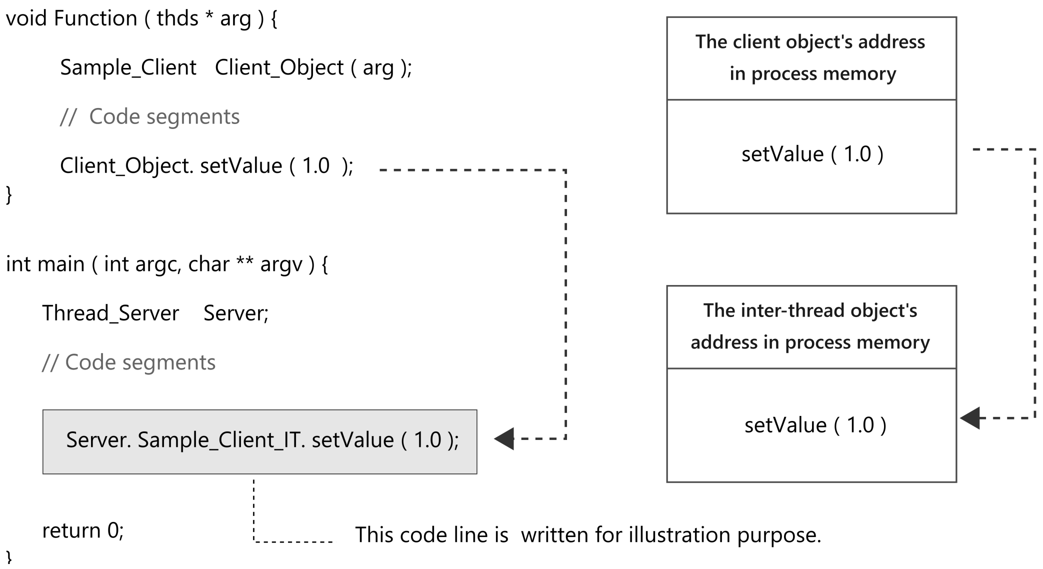
Thanks to the descriptor file, the pcynlitx platform knows the type of the objects that are used on the threads. Therefore, the client classes are written with a private pointer that points its corresponding inter-thread class. Then, on the construction of the client objects, the addresses of their corresponding inter-thread objects are received by the client object by means of thread-specific data structure (*thds*). It has been already indicated that the client objects are used as a connection point to the inter-thread objects. With the help of the client objects, the programmer can make indirection to the public member functions of the inter-thread objects without using any pointer. In order to achieve this, the client classes are build in a special form. The structure of the client classes is shown in below figure.

```
class INT {                                      class Client_INT {
public:                                          public:
   Type_1   MF_1 ( Pr_list_1 );  ········▶        Type_1   MF_1 ( Pr_list_1 );
   Type_2   MF_2 ( Pr_list_2 );  ········▶        Type_2   MF_2 ( Pr_list_2 );
       ⋮        ⋮          ⋮                          ⋮         ⋮          ⋮
   Type_n   MF_n ( Pr_list_n );  ········▶        Type_n   MF_n ( Pr_list_n );
private:                                         private:
                                                      INT *  Pointer ;
};                                               };
```

INT: Inter-Thread Class Type
Client_INT: The client of the inter-thread class INT
MF: An abbrevation for the term member function
Pr_list: An abbrevation for the term parameter list

Figure 6: The structure of the client classes

The declaration of a client class includes an exact copy of each public member function of its counterpart (*the corresponding inter-thread class*) and a private pointer member that points the objects which are in the type of its counterpart. On the inside of the member functions of the client classes, an indirection is performed directly to an instance of its counterpart. In other words, if a member function of a client object is called, in fact, the corresponding member function of the inter-thread object is called by means of the client object's pointer. A sample function call for the client objects is shown in below figure.



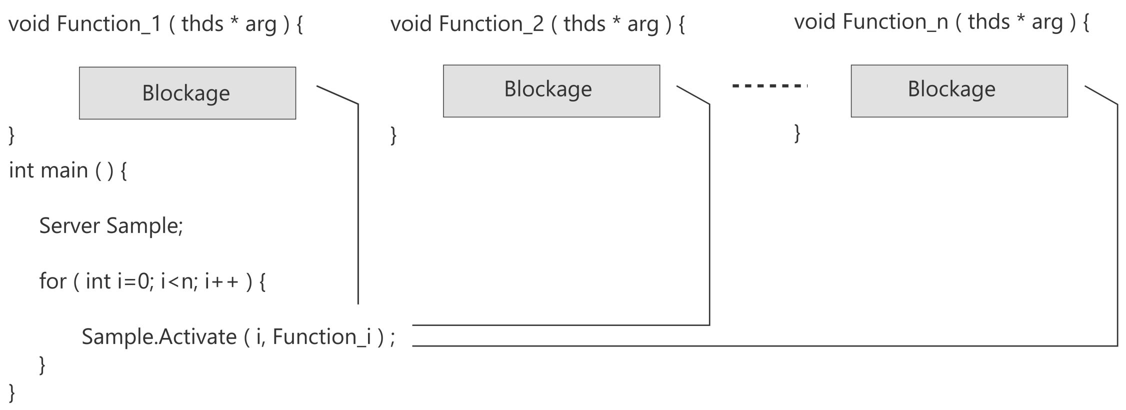
```
void Function ( thds * arg ) {

        Sample_Client   Client_Object ( arg );

        //  Code segments

        Client_Object. setValue ( 1.0  );
}

int main ( int argc, char ** argv ) {

      Thread_Server    Server;

      // Code segments

      Server. Sample_Client_IT. setValue ( 1.0 );


      return 0;
}
```

The client object's address in process memory
setValue (1.0)
The inter-thread object's address in process memory
setValue (1.0)
This code line is written for illustration purpose.

Figure 7: How the client objects are used

7 THE CONSTRUCTION OF THE MANAGER OBJECT AND ITS CLIENTS

In pcynlitx, as already mentioned, the name of the class that is used in order to synchronize the threads is Thread_Manager. This class is generated based on application specific data collected from the descriptor file. Beside to this, the manager object also collects some thread specific data from its clients on run-time. In fact, on each thread function, there must be a client of the manager object. On the instantiations of the clients of the manager object, the manager object receives not only the ID number of the thread but also the name of the thread function to be executed by the thread from its clients. As a result, before the synchronization tasks, the manager object knows exactly how many threads there are on the process, what the ID numbers of these threads are and which thread function is executed by which thread. In fact, the ID numbers of the threads are determined by the operating system on run-time. Therefore, in order to avoid some possible programming errors on the pcynlitx applications, it must be clearly understood that how the ID number of a thread is determined after the thread's creation on run-time. Actually, the instantiation of the clients of the manager object is performed on the beginning of the thread function (the first code line on the function routines). Just after the thread's creation, the threads call the constructors of the manager object's clients and the clients of the manager object makes indirection to the manager object's constructor at the same time. Then, all of the threads are blocked on inside of the manager object's constructor (the first line of the thread function routines) until each thread's data will be received by the manager object.



```
void Function_1 ( thds * arg ) {        void Function_2 ( thds * arg ) {        void Function_n ( thds * arg ) {

        Blockage                                Blockage                 ------         Blockage

}                                       }                                        }
int main ( ) {

    Server Sample;

    for ( int i=0; i<n; i++ ) {

          Sample.Activate ( i, Function_i ) ;
    }
}
```

Figure 8: The blockage on the manager object construction

In the figure, each thread executes its own thread function. Beside to this, each thread is blocked on the beginning of the thread function routine until the data of each thread will be received. However, this interaction between the manager object and its clients continues to the end of the process. in order to update its global knowledge about the process, the manager object continues to collect data from the process such as the status of the threads (*blocking* / *non-blocking*) if the threads continue their execution. In other words, the manager object updates its data autonomously after each operation that is performed on the process. The manager object is also an inter-thread object and as in the case of the other inter-thread objects, manager object is passed to the other threads over the server object. Of course, the client of the "Thread_Manager" class is named as "TM_Client" (*TM is an abbreviation of the Thread_Manager*). However, to be able to synchronize the threads, manager object can be reachable from each thread. Therefore, on each thread function, an instance of the TM_Client is required. Different from the other client classes, the constructor of the TM_Client takes two arguments. Its first argument is the address of thread specific data structure. The second argument is the name of the function that is executed by the thread and its type is standard C++ string. A sample illustration about the initialization of manager object is shown in below.

```
void Function( thds * arg ) {    // Thread Function

      TM_Client Manager ( arg,"procedure");
}
```

Figure 9: The manager object's instantiation

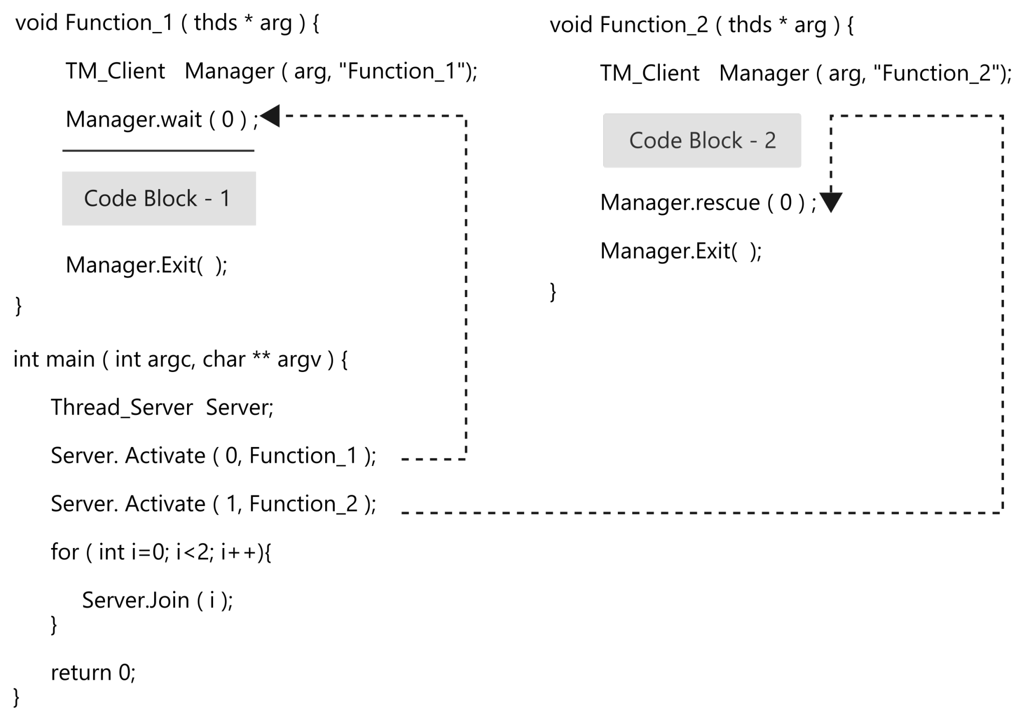
In order to prevent synchronization errors, the clients of the manager object (*the instances of the TM_Client*) must be instantiated on top of each thread function.

8 THREAD CONTROL FUNCTIONS

In the previous section, it has been indicated that the threads are controlled by means of the manager object member functions. Same of them have been listed in below.

8.1 Wait-rescue member functions with one parameter

When the thread specified on the argument performs a call to the "wait(int number)" member function, the caller thread is blocked. The same thread can be rescued by the corresponding rescue member function "rescue(int number)". A sample example for the usage of these member functions are given in below. In the figure, it can be easly seen that thread(0) waits before Code Block-1 until the thread (1) performs a call to the member function rescue(0) after execution of the Code Block-2. Therefore, the user of the pcynlitx can control the execution order of the threads.



```
void Function_1 ( thds * arg ) {

      TM_Client   Manager ( arg, "Function_1");

      Manager.wait ( 0 ) ;

      Code Block - 1

      Manager.Exit(  );
}

int main ( int argc, char ** argv ) {

     Thread_Server  Server;

     Server. Activate ( 0, Function_1 );

     Server. Activate ( 1, Function_2 );

     for ( int i=0; i<2; i++){

          Server.Join ( i );
     }

     return 0;
}
```

```
void Function_2 ( thds * arg ) {

      TM_Client   Manager ( arg, "Function_2");

      Code Block - 2

      Manager.rescue ( 0 ) ;

      Manager.Exit(  );
}
```

Figure 10: The wait member function with one argument

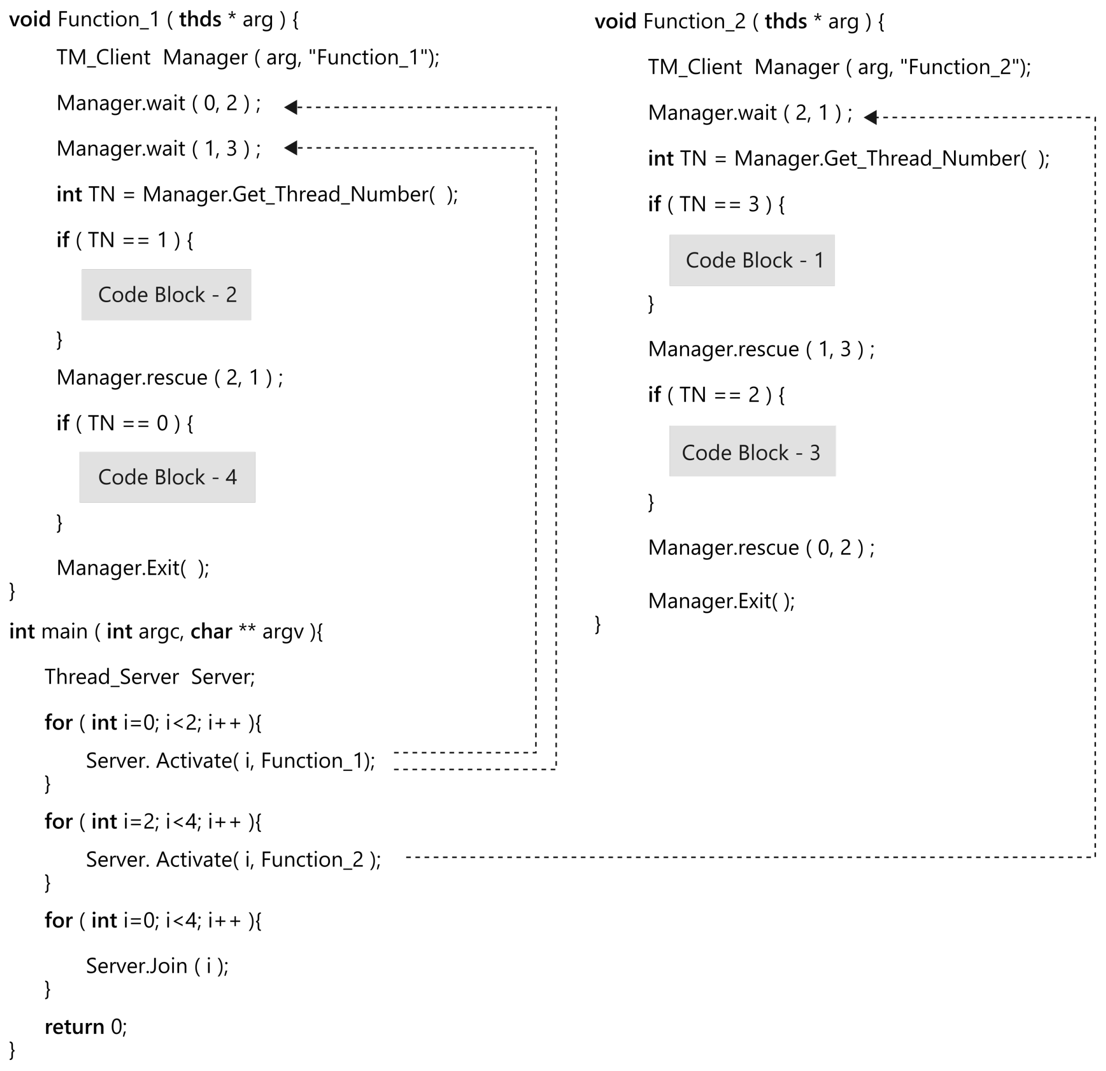
8.2 Wait-rescue member functions with two parameters

These member functions take two integer parameters. When the thread specified on the first parameter performs a call to the function, the caller thread is blocked. The second parameter is used in order to indicate the thread which will rescue the thread waiting later on. The corresponding rescue member function takes exactly same parameters and when the thread specified on the second parameter performs a call to the corresponding rescue member function, the thread waiting rescues from blockage. These member functions are given in below.

```
void wait ( int b, int r ) -> Blocks the thread - ( b )

void rescue ( int b, int r ) -> Rescues the thread - ( b )
```

Figure 11: The wait member function with two argument and related rescue function

let we consider that there are four different code section which must be executed in an order by different threads. An illustration of this thread control problem has been given in below figure. Now, let we assume that these code sections are named as Code Block-1, Code Block -2, Code Block -3 and Code Block-4. Then, let we also assume that the execution order of these code blocks are Code Block-1, Code Block -2, Code Block-3 and Code Block -4. Then,let we forward our assumption one step further and let we assume that the code blocked are written on two different function routines and each function routines are executed by two different threads. As you can see from the figure, the thread which is numbered with "0" (the thread(0)) executes the thread function named as "Function_1" and it is blocked when it performs a call to the function wait(0,2). Similarly, the thread (1) executes the same thread function and it is blocked when it performs a call to the function wait(1,3). Now, let we look at what is going on the other thread function. When the thread(2) performs a call to the function wait(2,3), it is blocked. However, for the thread numbered as "3" (the thread (3)) is not blocked in anywhere and thus, it continues its execution. Beside to this, in the figure, there is another control function which is shown as "Get_Thread_Number()". This control function returns the number of the caller thread.



```
void Function_1 ( thds * arg ) {
        TM_Client  Manager ( arg, "Function_1");
        Manager.wait ( 0, 2 ) ;
        Manager.wait ( 1, 3 ) ;
        int TN = Manager.Get_Thread_Number(  );
        if ( TN == 1 ) {
            Code Block - 2
        }
        Manager.rescue ( 2, 1 ) ;
        if ( TN == 0 ) {
            Code Block - 4
        }
        Manager.Exit(  );
}
int main ( int argc, char ** argv ){
      Thread_Server  Server;
      for ( int i=0; i<2; i++ ){
            Server. Activate( i, Function_1);
      }
      for ( int i=2; i<4; i++ ){
            Server. Activate( i, Function_2 );
      }
      for ( int i=0; i<4; i++ ){
            Server.Join ( i );
      }
      return 0;
}
```

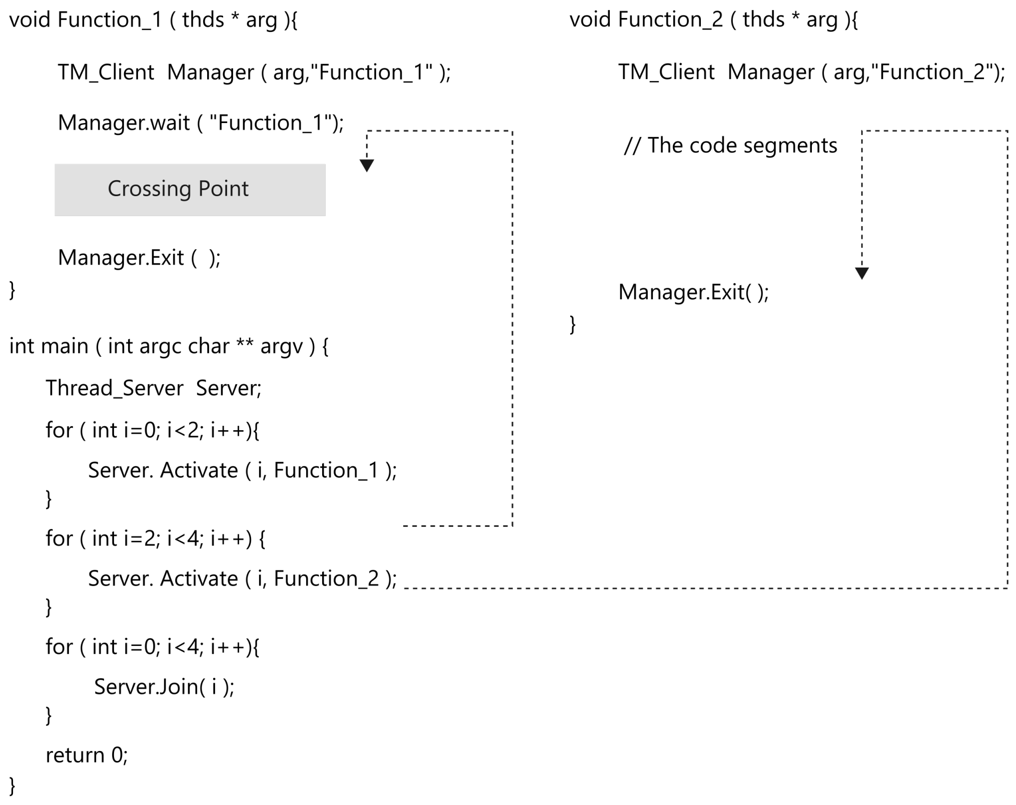
```
void Function_2 ( thds * arg ) {
        TM_Client  Manager ( arg, "Function_2");
        Manager.wait ( 2, 1 ) ;
        int TN = Manager.Get_Thread_Number(  );
        if ( TN == 3 ) {
            Code Block - 1
        }
        Manager.rescue ( 1, 3 ) ;
        if ( TN == 2 ) {
            Code Block - 3
        }
        Manager.rescue ( 0, 2 ) ;
        Manager.Exit( );
}
```

Figure 12: The wait member function with two arguments

In the figure, the integer variable which is shown as "TN" is used in order to hold the number of the threads. By this way, the inside of the if conditions are only executed by the thread indicated on the condition. This means that the Code Block-1 can only be executed by the thread(3). Beside to this, because of the other threads are in block condition, at first Code Block-1 is executed. Then, when the thread(3) performs a call to the function rescue(1,3), the thread(1) rescues from the block condition and it starts its execution. Therefore, secondly, the Code Block-2 is executed. Then, when it performs a call to the function rescue(2,1) which is written on the other thread function, the thread (2) rescues from the block condition and it starts its execution and therefore, the Code Block-3 is executed. Finally, when the thread(2) performs a call to the function rescue(0,2), the thread(0) rescues from the block condition and it starts its execution and the Code Block-4 is executed. This code example shows the power of the control tools that are used on the pcynlitx applications. The execution order and the priority of the threads reflect the term of scheduling. In the classical threading libraries, the scheduling of the threads are determined by the operating system and in each run of the process, the threads are scheduled differently. However, in pcynlitx applications, you can exactly control the execution order not only the threads but also particular code segments.

8.3 The blocking the threads executing a particular thread function

The pcynlitix provides unmatched control tools for thread synchronization. The one of them is blocking the threads executing particular thread function. If you want that the threads must pass from particular point of the code lines together, this member function of the manager object can be used. An example for the usage of wait member function blocking all of the threads executing the same thread function is shown in below figure.In the figure that is given in above, the threads executing the thread function which is named as "Function_1" are stopped until all of the threads executing the same thread function performs a call to the same member function - wait("Function_1"). In other words, the threads cross over the wait member function together.

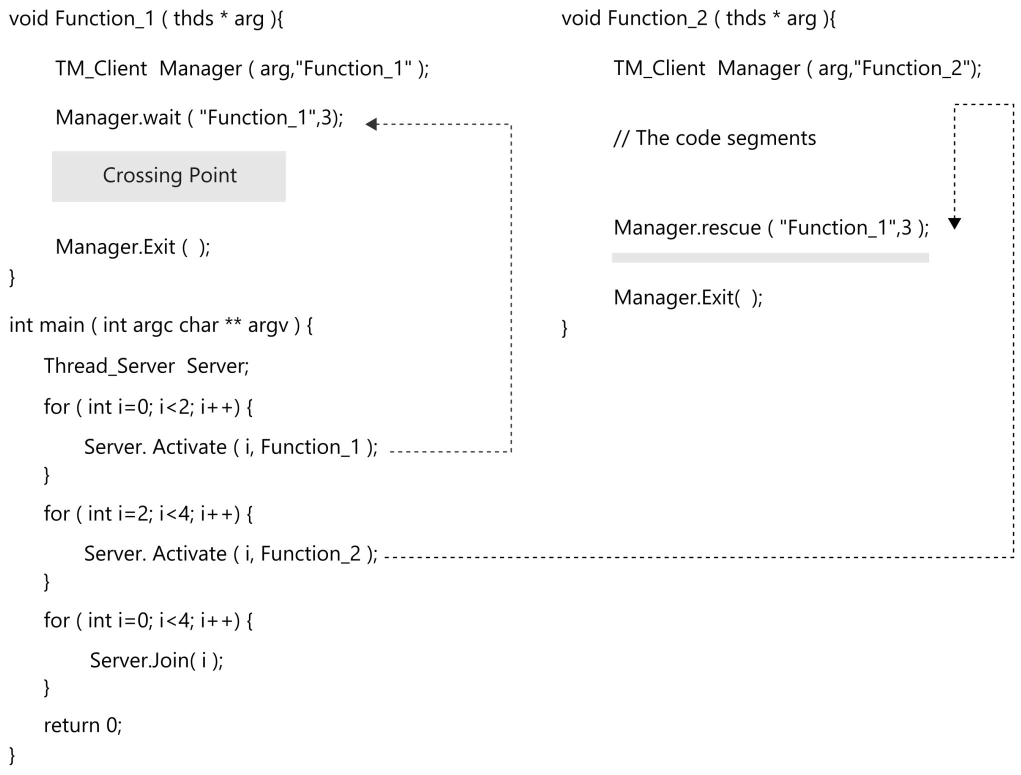


```
void Function_1 ( thds * arg ){

    TM_Client  Manager ( arg,"Function_1" );

    Manager.wait ( "Function_1");

    Crossing Point

    Manager.Exit (  );
}

int main ( int argc char ** argv ) {
   Thread_Server  Server;
   for ( int i=0; i<2; i++){
        Server. Activate ( i, Function_1 );
   }
   for ( int i=2; i<4; i++) {
        Server. Activate ( i, Function_2 );
   }
   for ( int i=0; i<4; i++){
         Server.Join( i );
   }
   return 0;
}
```

```
void Function_2 ( thds * arg ){

    TM_Client  Manager ( arg,"Function_2");

     // The code segments

    Manager.Exit( );
}
```

Figure 13: The wait member function acting as a crossing over point

There is another version of the same member function waiting a call to the its corresponding rescue member function form a thread executing a different thread function. An example that illustrates how this member function can be used is given in below.



```
void Function_1 ( thds * arg ){

    TM_Client  Manager ( arg,"Function_1" );

    Manager.wait ( "Function_1",3);

    Crossing Point

    Manager.Exit (  );
}

int main ( int argc char ** argv ) {
   Thread_Server  Server;
   for ( int i=0; i<2; i++) {
        Server. Activate ( i, Function_1 );
   }
   for ( int i=2; i<4; i++) {
        Server. Activate ( i, Function_2 );
   }
   for ( int i=0; i<4; i++) {
         Server.Join( i );
   }
   return 0;
}
```

```
void Function_2 ( thds * arg ){

    TM_Client  Manager ( arg,"Function_2");

    // The code segments

    Manager.rescue ( "Function_1",3 );

    Manager.Exit(  );
}
```

Figure 14: The wait member function blocking the execution of a thread function

8.4 Exit member function

In pcynlitx applications, at the end of each thread function, the usage of the member function which is named as "Exit( )" is mandatory. It informs the complation of the execution of the caller thread to the manager object.

8.5 The situation awareness in pcynlitx

In pcynlitx applications, you can get information about the situation of a particular thread in terms of blocked/non-blocked. In pcynlitx, the control function that is used for this purpose is "Get_Block_Status()" member function. The return type of the member function is boolean. The trace of the member function is given in below.

```
bool Get_Block_Status ( int number );
```

Figure 15: Getting block status of a thread

A simple example that illustrates how this member function can be used is given in the figure-16. The figure-16 is given on the next page. In the figure, an implementation of typical producer/consumer system has been depicted. In this typical example, the thread which is numbered as thread(0) reads a file and extracts some data from this file on the inside of a while loop. Then, it sends this extracted data into a buffer. Meanwhile, the thread which is numbered as thread(1) waits on the inside of the wait member function which is written on the inside of a do-while loop until the thread(0) sending a data into the buffer. After the data is sent into the buffer, the thread(0) sets the buffer empty condition as false and performs a call to the rescue(0) function. In this case, the thread(0) goes to the wait condition and thread(1) starts its execution. After that, thread(1) writes the collected data into a backup file and checks the status of the thread(0). If thread(0) is in the wait condition, then, thread(1) performs a call to the function rescue(0) and sets the buffer empty condition as true. This mechanism is very useful in order to prevent from the spurious wakeups. Moreover, this is not only benefit of retrieving status of the threads. In most cases, the errors that are faced on the multithread programming applications (*especially deadlocks*) occurs after a wrong rescue signal leading starting the execution of a thread before its time. Unfortunately, detection of this kind of errors is very hard. The logical understandability of the program to be developed can be listed as another benefits of this member function as well. However, in order to determine the status of the threads, the manager object have to collect information from the threads after each operation that is performed on the multi-threaded software. Therefore, the thread manager class has an array of boolean member holding the status of the threads. After each operation that is performed on the pcynlitx applications, the manager object updates its information about the status of the corresponding thread. For instance, on the inside of the wait and rescue member functions, the manager object's data members holding the status information the manager object is changed.

The pcynlitx platform provides more function retrieving information about the status of the threads. In pcynlitx, the other member function that retrieves information from the threads is "Get_Opt_Number( )". This member function returns the number indicating how many threads are running on the process currently.

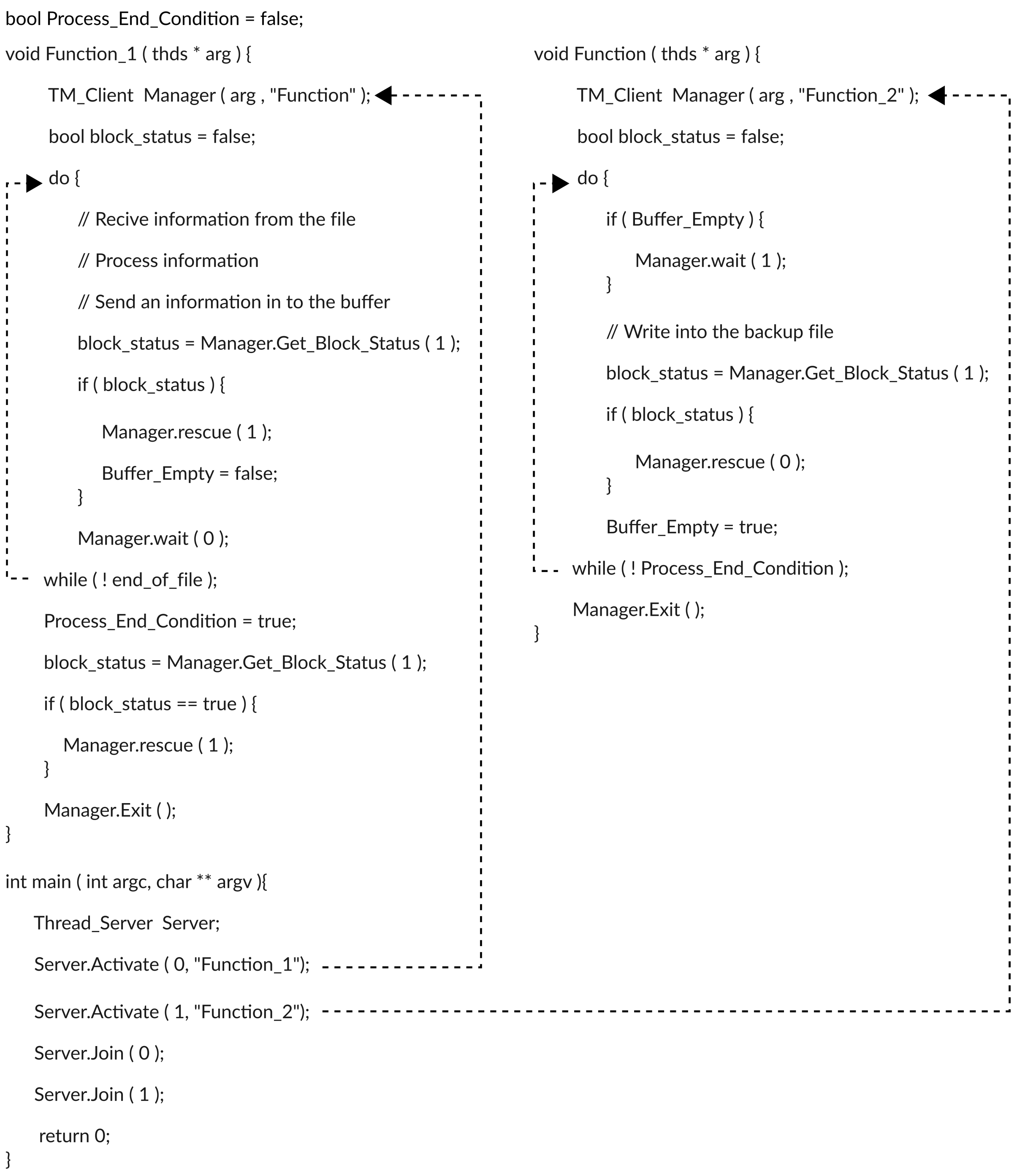
8.6 Stop a thread until a particular thread complates its exectuion

In pcynlitx, you can stop a certain thread until another thread's execution will be completed. This is another control function that can be used on the multithread applications in which pcynlitx platform is used. The name of this member function is "wait_until_exit" and it takes two parameters. The first one of them is the number of the thread that will be stopped and the other parameter is the number of the thread in which its completion is waited by.

```
void wait_until_exit ( int number_w, int number_c );

number_w : The number of the thread to be blocked until the thread

           specified as number_c complates its execution

number_c :  The number of the thread to be waited by
```

Figure 16: The member function stops thread to the end of other thread

An example for the usage of this member function can be find on the example codes given on the project repository and formal web site.



```
bool Process_End_Condition = false;
void Function_1 ( thds * arg ) {

       TM_Client  Manager ( arg , "Function" );

       bool block_status = false;

       do {

            // Recive information from the file

            // Process information

            // Send an information in to the buffer

            block_status = Manager.Get_Block_Status ( 1 );

            if ( block_status ) {

                Manager.rescue ( 1 );

                Buffer_Empty = false;
            }

            Manager.wait ( 0 );

      while ( ! end_of_file );

      Process_End_Condition = true;

      block_status = Manager.Get_Block_Status ( 1 );

      if ( block_status == true ) {

         Manager.rescue ( 1 );
      }

      Manager.Exit ( );
}

int main ( int argc, char ** argv ){

     Thread_Server  Server;

     Server.Activate ( 0, "Function_1");

     Server.Activate ( 1, "Function_2");

     Server.Join ( 0 );

     Server.Join ( 1 );

      return 0;
}
```

```
void Function ( thds * arg ) {

       TM_Client  Manager ( arg , "Function_2" );

       bool block_status = false;

       do {

            if ( Buffer_Empty ) {

                 Manager.wait ( 1 );
            }

            // Write into the backup file

            block_status = Manager.Get_Block_Status ( 1 );

            if ( block_status ) {

                 Manager.rescue ( 0 );
            }

            Buffer_Empty = true;

      while ( ! Process_End_Condition );

      Manager.Exit ( );
}
```

Figure 17: The member function giving the status of the threads