AYUDAME **Technical manual**

Steffen Brinkmann et al.

HLRS, Universität Stuttgart

Disclaimer: The information contained in this manual is not garanteed to be complete at this stage. It is subject to changes without further notice. Please send comments, corrections and additional text to temanejo@hlrs.de.

Steffen Brinkmann et al.:AYUDAME Technical Manual.

Contents

1 Introduction

AYUDAME is a generic library for communicating events occuring while running a StarSs task parallel application to the outside. The present version will set up a tcp/ip socket server, wait for a client to connect and send messages to it. A client designed to work with AYUDAME is TEMANEJO.

2 Quick start guides

2.1 For application programmers

In order to use AYUDAME you will have to compile the library. This is done with

shell: AYUDAME directory

```
# make && make install
```

This compiles the library and creates a link to it in the directory <ayudame_directory>/lib. If you encounter problems please read the README file in the AYUDAME directory and the comments in the Makefile.

To execute an application with AYUDAME enabled, run it as follows:

shell: AYUDAME directory

```
# LD_PRELOAD=<ayudame_directory>/lib/libayudame.so ./application <parameters>
```

The application will run until the start of the task parallel part. It will then pause and wait for a client (e.g. TEMANEJO) to connect. Then it will wait until a "step" request is issued by the client which is done by pressing the "Play" button in TEMANEJO.

2.2 For runtime programmers

In order to connect to programmes like TEMANEJO, the StarSs runtime has to call the `AYU_event` callback function. This function will send the appropriate message depending on the input parameters. Generally a call to `AYU_event` looks like this:

C example: call to AYU_event

```
#if USE_AYUDAME
        if (AYU_event) AYU_event(<event_id>, <task_id>, <void_data_pointer>);
#endif
```

The statement is enclosed in a preprocessor directive in order to be able to switch the AYUDAME support of when compiling the runtime. It is suggested to do so and to set `USE_AYUDAME` while configuring the runtime (when using autotools).

Furthermore the statement is only executed if an implementation of `AYU_event` is found at execution time. For details see sec. 4.

The `event_id` indicates which type of event is being communicated. It is an integer of type `enum ayu_event_t` declared in the header `Ayudame.h`:

C example: event id enum

```
enum ayu_event_t {
  AYU_EVENT_NULL = 0,
  AYU_PREINIT = 1,
  AYU_INIT = 2,
  AYU_FINISH = 3,
  AYU_REGISTERFUNCTION = 4,
  AYU_ADDTASK = 5,
  AYU_ADDHIDDENTASK = 6,
  AYU_ADDDEPENDENCY = 7,
  AYU_ADDTASKTOQUEUE = 8,
  AYU_PRESELECTTASK = 9,
  AYU_PRERUNTASK = 10,
  AYU_RUNTASK = 11,
  AYU_POSTRUNTASK = 12,
  AYU_RUNTASKFAILED = 13,
  AYU_REMOVETASK = 14,
  AYU_WAITON = 15,
  AYU_BARRIER = 16,
  AYU_ADDWAITONTASK = 17
};
```

Not all of these events are necessary. A minimalistic sequence of events could look like this:

C example: minimalistic example

```
1 #include <Ayudame.h>
2 ayu_runtime_t ayu_rt = AYU_RT_OMPSS; //for OMPSs runtime
3 AYU_event(AYU_PREINIT, 0, (void*) &ayu_rt );
4 int64_t AYU_data1[2] = { function_id_1, task_is_critical_1 };
5 AYU_event(AYU_ADDTASK, 1, AYU_data1);
6 int64_t AYU_data2[2] = { function_id_2, task_is_critical_2 };
7 AYU_event(AYU_ADDTASK, 2, AYU_data2);
8 uintptr_t Ayu_data3[3] = { 1, mem_address, original_mem_address };
9 AYU_event(AYU_ADDDEPENDENCY, 2, AYU_data3);
```

Please note that `Ayudame.h` has to be in the include path and that for easier readability the checks of the form `if (AYU_event)` were omitted. Thus this code will only work if libayudame.so is preloaded.

Let us step through the example above. In line 1 the header `Ayudame.h` is included. This is necessary for providing the declarations of the callback functions and the event enum type. In line 2 a variable of type `ayu_runtime_t` is declared and defined to hold an identifier for the runtime. These identifiers are declared in `Ayudame.h` (see Appendix). In line 3 `AYU_event` is called for the first time with the event id `AYU_PREINIT`. When this event is sent, Ayudame will set up the socket server and wait for a client to connect. The port of the socket connection is set by the environment variable `AYU_PORT`. If this

variable is not defined, the port is set to

$$\text{port} = \text{pid} \,\%\, 1000 + 5000,$$

i.e. for a process id of 23674, the port will result in 5674.

In line 4 the additional data is set up for adding a task to the dependency graph. It consists of the function id of the task and an integer indicating whether the task is to be executed with priority (1) or not (0).

In the next line (4) the creation of the task is communicated to AYUDAME. The function parameters are the type of the event (`AYU_ADDTASK`), the task id (1) and the additional data. We repeat the last to steps for another task with the task id 2.

Finally we define a dependency between these tasks in lines 8 and 9. Here the additional data consists of the task id of the task which "causes" the dependency, the memory address of the dependency and (in the case of renaming, see SMPSs manual) the original memory address.

The call to `AYU_event` contains the type of event `AYU_ADDDEPENDENCY`, the task id of the task "suffering" the dependency and the additional data.

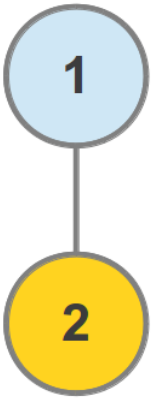

Figure 1: minimalistic example graph

This piece of code will result in a dependency graph shown in figure 1.

3 Events and requests

Genrally speaking the interactions of AYUDAME with the world can be divided into two types: **events** and **requests**. Events represent the information flow from the application and the runtime environment to the outside world (e.g. a file, a logging tool or a debugger) while requests are passed from the outside world to the application.

The events are handled runtime-independant by the `AYU_event()` callback function. Requests on the other hand are highly runtime dependant. The set of available request varies strongly from one runtime to the other.

3.1 Events

Events are communicated by the use of the `AYU_event()` callback function. It is declared in `Ayudame.h` as follows:

```
1 void AYU_event(ayu_event_t event, const int64_t taskId, void *p)
2     __attribute__ ((weak));
```

This declaration permits to leave the header included even if the library `libayudame.so` is not linked or preloaded. In that case the callback name `AYU_event` is equivalent to `false`. Therefore it is recommended to call this function within a conditional clause:

```
1 if ( AYU_event ) AYU_event(AYU_ADDTASK, 1, Ayu_data);
```

The parameters passed to `AYU_event` consist in an event type, a task id and additional data the content of which depends on the event type. Three events are mandatory for building the graph: `AYU_PREINIT`, `AYU_ADDTASK` and `AYU_ADDDEPENDENCY`.

Find a comprehensive list of events in following:

`AYU_EVENT_NULL`: dummy event

optional for initialisation of event variables, event Id: ***0***

Does not represent an event.

`AYU_PREINIT`: Pre-initialisation

mandatory for logging, event Id: ***1***

Should latest be emmited just before the parallel execution starts. Must be the first event to be emitted.

The additional data consists of two items. The first is an integer denoting which runtime is in use: CppSs (1), SMPSs (2), OMPSs (3), GOMP (4) or (CILK), the second parameter is optional and consists of the integer process id (PID).

```
uint64_t AYU_data[2] = {(uint64_t) AYU_RT_SMPSS, (uint64_t) getpid()};
if (AYU_event) { AYU_event(AYU_PREINIT, 0, (void*) AYU_data ); }
```

`AYU_INIT`: Initialisation

optional for logging, event Id: ***2***

Should be sent just after `AYU_PREINIT`. The execution will pause if pausing is available for the specified runtime until a request is received in order to proceed to execute the parallel part of the application.

```
if (AYU_event) { AYU_event(AYU_INIT, 0, NULL); }
```

`AYU_FINISH`: Finalisation

recommended for closing socket connection, event Id: ***3***

This event notifies that the parallel part of the application has finished. The socket connection will be closed.

```
if (AYU_event) {AYU_event(AYU_FINISH, 0, NULL); }
```

`AYU_REGISTERFUNCTION`: Register taskified functions

optional for function name display, event Id: ***4***

This event registers the function name and the function Id of a taskified function.

```
if (AYU_event) { AYU_event(AYU_REGISTERFUNCTION, id, (void *)name); }
```

`AYU_ADDTASK`: Create task

mandatory for graph display, event Id: ***5***

A task is represented by a node in the dependency graph. When a task is created a notification should be emitted containing the task Id and the function Id of the function executed by the task. It is recommended to either make sure that every function is registered *before* the first task for this function is created, or *not* to register function names at all.

The additional data must be a pointer to an integer denoting whether the task is critical (high-priority, =1) or not (=0).

```
int64_t AYU_data[2] = {functionId,(int)task->isCritical()};
if (AYU_event) { AYU_event(AYU_ADDTASK, task->getId(), AYU_data); }
```

`AYU_ADDHIDDENTASK`: Add hidden task

do not use, development in progress, event Id: ***6***

Do not use! This is under development!

`AYU_ADDDEPENDENCY`: Define dependency

mandatory for graph display, event Id: ***7***

A dependency is represented by an edge in the dependency graph. When a dependency is created, the application should notify that passing along the task Id of the predecessor, the memory address of the variable or array causing the dependency and the original memory address of the variable (in case of renaming). The address can be 0 if unknown.

It is necessary that both tasks (the dependant and the task it depends on) are already created when defining a dependency.

```
uintptr_t Ayu_data[3]={0,0,0};
Ayu_data[0] = otherTask->getId();
Ayu_data[1] = mem_addr_of_dependecy;
Ayu_data[2] = original_mem_addr_of_dependecy;
if (AYU_event) {
  AYU_event(AYU_ADDDEPENDENCY, task->getId(), (void*) Ayu_data);
}
```

`AYU_ADDTASKTOQUEUE`: Queue task notification

recommended for graph display, event Id: ***8***

When a task does not depend on any other tasks it can be marked as queued. The task must exist prior to this event. Additionally the Id of the thread which queued the task can be passed.

```
intptr_t thread_id=getThreadId();
if (AYU_event) {
```

```
  AYU_event(AYU_ADDTASKTOQUEUE, task->getId(), &thread_id);
}
```

`AYU_PRESELECTTASK`: **Task is about to be dequeued**
do not use, debugging event, event Id: ***9***

This event notifies that the runtime is checking the queue(s) for a task to run. As this event will usually occur many times, it is completely ignored, i.e. not even logged.

`AYU_PRERUNTASK`: **Prerun task notification**
recommended for graph display, event Id: ***10***

When a task is scheduled for execution, i.e. an execution thread has dequeued the task it can be marked as "about to run". This event is recommended to be emitted before pausing the execution when stepping through the dependency graph.

The number of nodes marked this way is at any time equal or less the number of execution threads. Additionally the Id of the thread which runs the task can be passed. Note, that the thread Id needs to be non-negative here.

```
intptr_t thread_id=getThreadId();
if (AYU_event) {
  AYU_event(AYU_PRERUNTASK, task->getId(), &thread_id);
}
```

`AYU_RUNTASK`: **Run task notification**
optional for time measurement, event Id: ***11***

Just before the task is executed a "run task" event can be emitted. This event is intended for time measurement. Between this event and the actual execution of the task there should be as less code as possible to ensure exact time stamps.

```
if (AYU_event)  { AYU_event(AYU_RUNTASK, task->getId(), NULL); }
```

`AYU_POSTRUNTASK`: **Task execution finished**
optional for logging, event Id: ***12***

```
if (AYU_event) { AYU_event(AYU_POSTRUNTASK, task->getId(), NULL); }
```

`AYU_RUNTASKFAILED`: **Task execution failed**
optional for failed tasks, event Id: ***13***

```
if (AYU_event) { AYU_event(AYU_RUNTASKFAILED, task->getId(), NULL); }
```

`AYU_REMOVETASK`: **Task execution finished**
recommended for graph display and time measurement, event Id: ***14***

When a task has finished this event should be emitted.

```
if (AYU_event) { AYU_event(AYU_REMOVETASK, taskNode->getId(), NULL); }
```

`AYU_WAITON`: **Wait-on event**
do not use, development in progress, event Id: ***15***

For treating wait-on events.
Do not use! This is under development!

`AYU_BARRIER`: **Barrier notification**
recommended for graph display with horizontal barrier lines, event Id: ***16***

Notifies of a barrier.

```
if (AYU_event) { AYU_event(AYU_BARRIER, 0,NULL); }
```

`AYU_ADDWAITONTASK`: **Wait-on event**
do not use, development in progress, event Id: ***17***

For treating wait-on events.
Do not use! This is under development!

3.2 Requests

tbd.

4 The callback functions

These functions are declared in the header file `Ayudame.h` as weak references. This way it is possible to check whether an implementation of the functions exist during runtime. This enables an application to run with or without the preloaded AYUDAME library without recompiling (see code example in sec. 2.2).

4.1 The `AYU_event` callback function

tbd.

A Sources

A.1 The header `Ayudame.h`

Listing 1: The main header of the Ayudame package

```
/****H* Ayu/Ayudame.h
 *  NAME
 *    Ayudame.h - header file containing the event and request types of the
 *    Ayudame package and callback function declarations.
 *  DESCRIPTION
 *    This header file declares the following enumeration types:
 *    * ayu_event_t
 *    * ayu_request_t
 *    * ayu_runtime_t
 *    And these function declarations:
 *    * AYU_event
 *    * AYU_getBreakpoint
 *    * AYU_getNumThreads
 *    * AYU_getPriorityLevel
 *    * AYU_isBlocked
 *    * AYU_registerTask
 *    * AYU_setNumThreads
 *    And these global constants:
 *    * AYU_buf_size
 *    * AYU_string_buf_size
 *  COPYRIGHT
 *    (C) 2010-2013 HLRS, University of Stuttgart
 *    Ayudame is published under the terms of the BSD license.
******/

#ifdef SWIG
%{
#include "Ayudame.h"
%}
// SWIG does not understand __attribute__, so unset it.
#define __attribute__(x)
#endif

#ifndef Ayudame_H
#define Ayudame_H

#define AYU_API_VERSION 1

#include <stdint.h>
#include <unistd.h>

#ifndef AYU_MASTER_TASKID
#  define AYU_MASTER_TASKID 0
#endif

#ifdef __cplusplus
extern "C" {
#endif

/*****t* Ayudame/ayu_event_t
 *  NAME
 *    ayu_event_t - enum of events
 *  DESCRIPTION
 *    ayu_event_t consists of the following events:
 *    *  AYU_EVENT_NULL -- "no event", used for initialisation etc.
 *    *  AYU_PREINIT -- establish socket connection. This should be the
 *       first event to be issued. It will notify Ayudame of the Runtime type.
 *    *  AYU_INIT -- can be issued after runtime initialisation. Can contain
 *       the number of processing resources. Temanejo will pause the program
 *       when AYU_INIT is issued by default.
 *    *  AYU_FINISH -- notify of the end of the parallel  execution.
```

```
 *     *  AYU_REGISTERFUNCTION -- can be issued to pass the function name of a
 *        taskified function.
 *     *  AYU_ADDTASK -- to notify of the creation of a task instance.
 *     *  AYU_ADDHIDDENTASK -- deprecated
 *     *  AYU_ADDDEPENDENCY -- notify of a dependecy between two tasks. Can
 *        contain two ids (e.g. memory addresses) to dispay coloures edges in
 *        Temanejo.
 *     *  AYU_ADDTASKTOQUEUE -- notify of queueing a task.
 *     *  AYU_PRESELECTTASK -- deprecated
 *     *  AYU_PRERUNTASK -- notify that a task is dequeued and assigned to
 *        a precessing resource.
 *     *  AYU_RUNTASK -- to be issued just before a task is executed. For time
 *        measurement.
 *     *  AYU_POSTRUNTASK -- to be issued right after task finished. For time
 *        measurement.
 *     *  AYU_RUNTASKFAILED -- notify of failure to run task
 *     *  AYU_REMOVETASK -- notify of successfully finished task. Removed from
 *        the graph.
 *     *  AYU_WAITON -- notify of "wait-on" event
 *     *  AYU_BARRIER -- notify of "barrier" event
 *     *  AYU_ADDWAITONTASK -- deprecated
 *
 *   SOURCE
 */

typedef enum ayu_event_t {
  AYU_EVENT_NULL = 0,
  AYU_PREINIT = 1,
  AYU_INIT = 2,
  AYU_FINISH = 3,
  AYU_REGISTERFUNCTION = 4,
  AYU_ADDTASK = 5,
  AYU_ADDHIDDENTASK = 6,
  AYU_ADDDEPENDENCY = 7,
  AYU_ADDTASKTOQUEUE = 8,
  AYU_PRESELECTTASK = 9,
  AYU_PRERUNTASK = 10,
  AYU_RUNTASK = 11,
  AYU_POSTRUNTASK = 12,
  AYU_RUNTASKFAILED = 13,
  AYU_REMOVETASK = 14,
  AYU_WAITON = 15,
  AYU_BARRIER = 16,
  AYU_ADDWAITONTASK = 17
} ayu_event_t;

/******/

/****t* Ayudame_types.h/ayu_request_t
 *   NAME
 *     ayu_request_t - enum of requests
 *   DESCRIPTION
 *     ayu_event_t consists of the following events:
 *     * AYU_REQUEST_NULL -- "no request", used for initialisation
 *     * AYU_NOREQUEST -- explicit "no request"
 *     * AYU_PAUSEONEVENT -- pause request, event upon which the program should
 *       pause is given as third parameter, fourth parameter is 0 for "off"
 *       ("un-pause") and 1 for "on"
 *     * AYU_PAUSEONTASK --  pause request, task id is given as third parameter,
 *       fourth parameter is 0 for "off" ("un-pause") and 1 for "on"
 *     * AYU_PAUSEONFUNCTION -- pause request, function id is given as third
 *       parameter, fourth parameter is 0 for "off" ("un-pause") and 1 for "on"
```

```
 *     * AYU_STEP -- run until next pause condition is reached
 *     * AYU_BREAKPOINT -- set breakpoint, i.e. don't assign new tasks, third
 *       parameter is 0 for "off" ("un-pause") and 1 for "on"
 *     * AYU_BLOCKTSK -- to block a specific task. Task id has to be passed.
 *     * AYU_PRIORITISETASK -- to set the priority level of a specific task.
 *     * AYU_SETNUMTHREADS -- to set the number of processing resources.
 *   SOURCE
 */

typedef enum ayu_request_t{
  AYU_REQUEST_NULL = 0,
  AYU_NOREQUEST = 1,
  AYU_PAUSEONEVENT = 2,
  AYU_PAUSEONTASK = 3,
  AYU_PAUSEONFUNCTION = 4,
  AYU_STEP = 5,
  AYU_BREAKPOINT = 6,
  AYU_BLOCKTASK = 7,
  AYU_PRIORITISETASK = 8,
  AYU_SETNUMTHREADS = 9
} ayu_request_t;
/******/

/****t* Ayudame_types.h/ayu_runtime_t
 *   NAME
 *     ayu_runtime_t - enum of runtime ids
 *   DESCRIPTION
 *     ayu_runtime_t consists of the following ids:
 *     * AYU_RT_UNKNOWN -- no or unknown runtime, used for initialisation
 *     * AYU_RT_CPPSS -- CPPSS runtime
 *     * AYU_RT_SMPSS -- SmpSs runtime
 *     * AYU_RT_OMPSS -- OmpSs runtime
 *     * AYU_RT_STARPU -- StarPU runtime
 *     * AYU_RT_FASTFLOW -- FastFlow runtime
 *     * AYU_RT_MPI --  MPI runtime
 *     * AYU_RT_GOMP -- GNU OpenMP runtime
 *     * AYU_RT_CILK -- CILK runtime
 *   SOURCE
 */
typedef enum ayu_runtime_t{
  AYU_RT_UNKNOWN = 0,
  AYU_RT_CPPSS = 1,
  AYU_RT_SMPSS = 2,
  AYU_RT_OMPSS = 3,
  AYU_RT_STARPU = 4,
  AYU_RT_FASTFLOW = 5,
  AYU_RT_MPI = 6,
  AYU_RT_GOMP = 7,
  AYU_RT_CILK = 8
} ayu_runtime_t;
/******/

  /****f* Ayudame/AYU_event
   *   NAME
   *     AYU_event
   *     - callback for the *Ss runtime
   *   SYNOPSIS
   *     AYU_event(event, taskId, pointer)
   *   DESCRIPTION
   *     this function is called from the task parallel runtime to communicate
   *     with the Ayudame package. Events of type ayu_event_t, the task ID
   *     of the calling task and a pointer to additional data can be sent.
```

```
   *    AYU_event will connect to a client via tcp sockets and pass on
   *    the information.
   *  PARAMETERS
   *    ayu_event_t event       // type of event
   *    const int64_t taskId    // taskId from which the event is sent
   *    void *p                 // pointer to further data
   *  RETURN VALUE
   *    void
   *  SOURCE
   */
#ifndef SWIG
  void AYU_event(ayu_event_t event, const int64_t taskId, void *p)
    __attribute__ ((weak));
#endif
  /******/

  /****f* Ayudame/AYU_getBreakpoint
   *  NAME
   *    AYU_getBreakpoint
   *    - return AYU_breakpoint
   *  SYNOPSIS
   *    breakpoint = AYU_getBreakpoint()
   *  DESCRIPTION
   *    Locks the global variable AYU_breakpoint, returns its value
   *    and unlocks it
   *  PARAMETERS
   *    void
   *  RETURN VALUE
   *    1 if AYU_breakpoint is set
   *    0 if AYU_breakpoint is not set
   *  SOURCE
   */
#ifndef SWIG
  int AYU_getBreakpoint(void) __attribute__ ((weak));
#endif
  /******/

  /****f* Ayudame/AYU_isBlocked
   *  NAME
   *    AYU_isBlocked
   *    - return true if task is blocked
   *  SYNOPSIS
   *    if(AYU_isBlocked(taskId)) { do_something(); }
   *  DESCRIPTION
   *    Locks the global variable AYU_blockedTask using the
   *    breakpoint mutex lock, compares to it
   *    taskId given as parameter and returns result
   *  PARAMETERS
   *    * const int taskId
   *  RETURN VALUE
   *    true if task is blocked
   *    false if task is not blocked
   *  SOURCE
   */
#ifndef SWIG
  unsigned AYU_isBlocked(const int taskId) __attribute__ ((weak));
#endif
  /******/

  /****f* Ayudame/AYU_getPriorityLevel
   *  NAME
   *    AYU_getPriorityLevel
```

```
    *    - returns priority of task
    *  SYNOPSIS
    *    AYU_getPriorityLevel(taskId)
    *  DESCRIPTION
    *    returns the priority of a task identified by its id.
    *    If the priority has not been changed in Ayudame it
    *    returns -1.
    *  PARAMETERS
    *    int taskId
    *  RETURN VALUE
    *    -1: if priority has not changed
    *    0:  low (no) priority
    *    positive integer: priority level
    *  SOURCE
    */
#ifndef SWIG
  int AYU_getPriorityLevel(int taskId) __attribute__ ((weak));
#endif
  /******/

  /****f* Ayudame/AYU_getNumThreads
    *  NAME
    *    AYU_getNumThreads
    *    - returns number of threads set by Ayudame
    *  SYNOPSIS
    *    runtime_setNumThreads(AYU_getNumThreads());
    *  DESCRIPTION
    *    returns the number of threads as set by ayudame
    *  PARAMETERS
    *    void
    *  RETURN VALUE
    *    unsigned int: number of threads
    *  SOURCE
    */
#ifndef SWIG
  unsigned int AYU_getNumThreads() __attribute__ ((weak));
#endif
  /******/

  /****f* Ayudame/AYU_setNumThreads
    *  NAME
    *    AYU_setNumThreads
    *    - sets number of threads in the runtime
    *  SYNOPSIS
    *    AYU_setNumThreads(n_threads);
    *  DESCRIPTION
    *    This callback function can be implemented in the StarSs
    *    runtime. If it is not implemented, the client of the
    *    Ayudame library will not be able to change the number of
    *    threads.
    *  PARAMETERS
    *    unsigned int n_threads
    *  RETURN VALUE
    *    void
    */
#ifndef SWIG
  void AYU_setNumThreads(unsigned int n_threads) __attribute__ ((weak));
#endif
  /******/


  /****f* Ayudame/AYU_registerTask
```

```
    *  NAME
    *     AYU_registerTask
    *     - obsolete. kept for backward compatibility
    *  SYNOPSIS
    *
    *  DESCRIPTION
    *
    *  PARAMETERS
    *
    *  RETURN VALUE
    *
    *  SOURCE
    */
#ifndef SWIG
   void AYU_registerTask(void*) __attribute__ ((weak));
#endif
   /******/

#ifdef __cplusplus
}
#endif

#endif //Ayudame_H
```