

# GMIS User Manual

Hongbing Zhang [hongbing75@gmail.com](mailto:hongbing75@gmail.com)

Copyright @2002-2014 All rights reserved

## Introduction

General Machine Intelligent System is designed to build a general used robot brain, which can convert all the machines with modern operating system, such as PC, cell phone, smart TV, intelligent vehicle and other industrial machineries, into a robot that can be actuated by ordinary people with their native language.

Different from existing Intelligent Voice Assist, GMIS supports a quasi natural language that can compatible with the users' mother tone and through which the users can control robot's real-time behavior easily. GMIS brings practicability and flexibility for the application of robots.

We can already order a GMIS prototype with quasi natural language to play online Texas Poker autonomously. It will show us these abilities of GMIS:

1. Can be driven by quasi natural language programming  
All the programming languages are designed for the programmers so far and this severely limited the interactions between ordinary people and the machine. Those old people, especially who can only use their mother tongue and need the robot's help to take care of them very much, are badly in need of a robot that can be driven by natural language. Nowadays, we have a language called Final C to help users communicate with GMIS prototype and it actually the extremely simplified version of natural language. The users don't need to study more and Final C can help GMIS create practical value in the near future. We will work on prompting GMIS's understanding ability for natural language step by step.
2. Realize the diversity of robots with instincts  
GMIS provides all the robots a uniform and configurable intelligent structure, through which different robots can get a specialized instinct according to their design purposes, and finally realize the diversity of robots.
3. Learn and memorize like human beings  
Just like an infant, robot's languages and abilities are acquired. Robots with same instinct can gain different abilities as they have different learning experiences. That means users need to teach their robots and as you can image.
4. Actuating logic like neural tree of human beings

GMIS will compile all the task logic into a tree to implement. It is the key for GMIS to program and think autonomously in the future that a tree will still be a tree after reorganization. It's also the first time to realize this kind of design in the world.

5. The ability of logically retrieve

Our human beings can temporarily stop a task after half done without occupy the brain and continue the task after years instead of restart. Similarly, GMIS can also temporarily stop a logic task, release memory and even the computer was shut down, it can still continue the task after the computer restarted.

6. Realize dynamic logic

Just like human beings can change their scheduled actions as the conditions changed, GMIS can also actuate logic while building it, or change an actuating logic during operation.

7. Can use external objects

Through external objects, GMIS can perfectly implement the existing application code in a touchable manner, which also has an essential practical significance.

8. A nature of parallel implementation ability

Parallel computing has always been a difficulty and hot issues especially in the situation that that where parallel and serial is mixed. It's a basic ability for human brain but there still have no all-purpose programming pattern. We usually say: "first get A and B done at the same time, then C", that is a cooperative computing for parallel and serial. Now GMIS can also do this easily.

9. Specialization and cooperation with other robots

Just like human beings, the robot based on GMIS can all realize interconnection and interworking and they can constitute dynamically into a web of things, do the logical tasks by specialization and cooperation and release an enormous productivity.

Actually, you can regard GMIS as a next-generation operating system and it will finally cover the existing kind and make it become a basement system.

This text is aimed at helping customers know the feature and breakthroughs in technology as mentioned above and provide a series of practical application case to show the practical value of GMIS prototype.

Following this manual, you can operate the matched GMIS prototype to test its functions in your PC (windows only so far). Clearly, without strict test, GMIS prototype may have all kinds of BUGs and you'd better have some C++ programming experience.

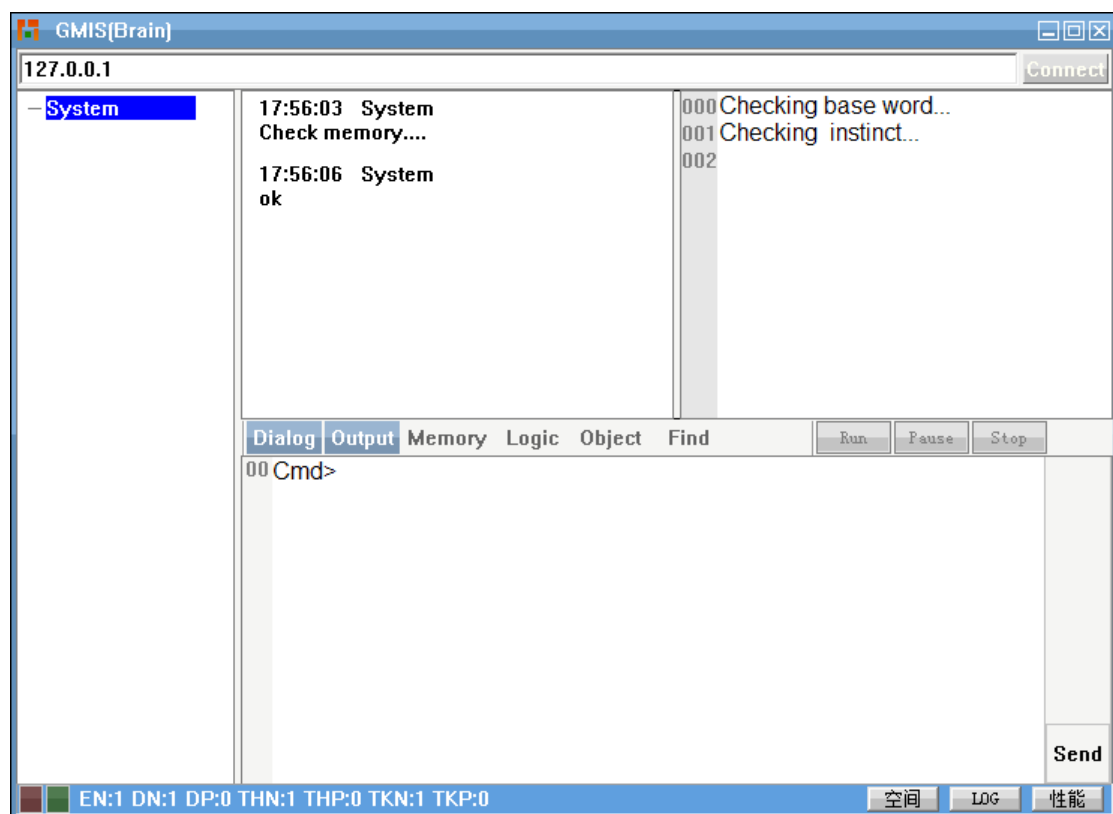
# User Interface

Now let's have a look at GMIS. An entire GMIS is consisted by three parts, or in other word three programs, which represent: brain, space entry and actuator.

As we have no organ to provide information and feedbacks to the brain, we developed a digital space and the brain can directly use objects in it. Actuator's work is to simulate the physical reactions, which in fact is to drive the objects that are in the digital space.

## Brain

Theoretically, there should have a message organ to provide in-out to the brain. Now we use a UI to do it. This UI can remote connect the brain and certainly it can also integrate with the brain at local. If start GMIS.exe in the installation directory, the user can see:



As the shown picture, there are three parts in the UI: A dialogue list in the left, an input box in low right, and a message output window in upper right.

### Dialogue list

There is a tree view on the left of the UI and the dialogue happening in the brain will be

shown here. Each dialogue means one thought or one task in the brain.

The system will provide a default main dialogue. Main dialogue is like the nerve center and all information from outside will be sent to it first. It will generate a sub-dialogue if needed and send the message to the sub-dialogue for implementation.

Sub-dialogue can also have its sub-dialogue and every sub-dialogue serves its parent dialogue. The implementation result will be feed backed to its own parent dialogue. This process is like when you are doing a task, you have a question to ask your workmate, thus a sub-dialogue generated. After your workmate gives you the answer, you go back to what you were doing before. In another word, GMIS has the ability of doing interactive logic dialogue.

### **Input box**

There's an input box in low right, and the user can click a dialogue in the dialogue list to select it as the focused one. Just input messages in the input box, then click "Send" button to send the message to the focus dialogue for implementation.

### **Output window**

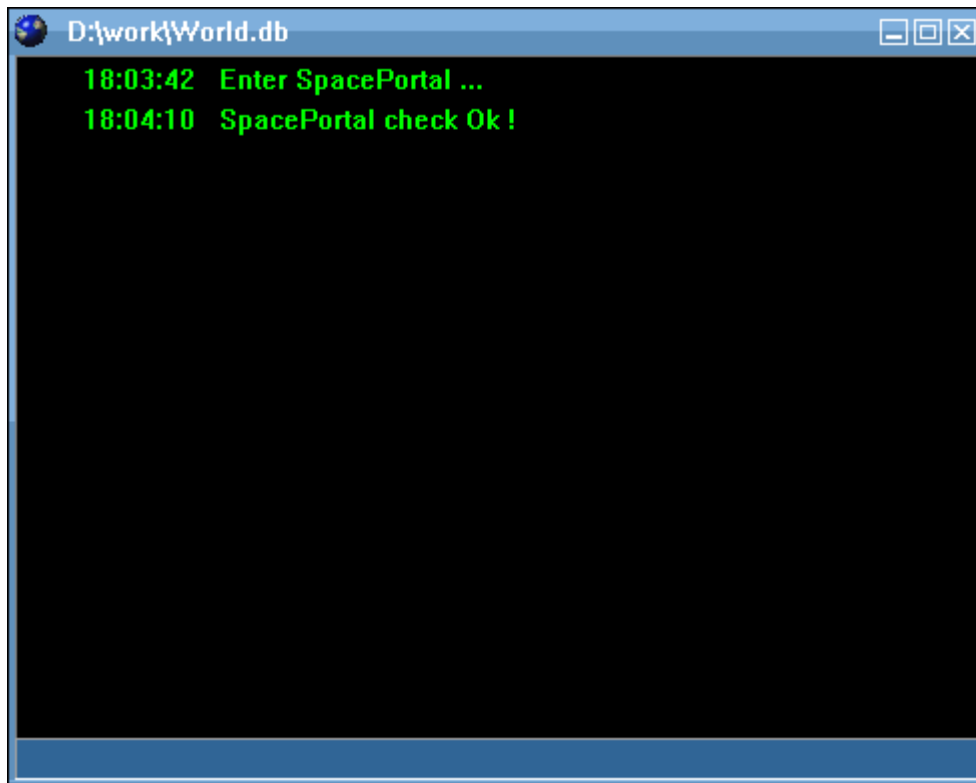
Output window has many sub-windows that can show different messages. Users can open or close the windows by clicking buttons in toolbar in the middle.

Mainly includes:

- 1) Dialog view: Display the dialog record of the user as well as the feedbacks.
- 2) Output view: Display the output information of the logic while implementation.
- 3) Memory view: Display temporary memory, in other words ephemeral data that generated as the current dialog logic implemented. These ephemeral data is only valid for the current dialog.
- 4) Object view: Actually it's also a temporary memory window and it specifically responsible for the current used exterior objects. And we will explain what is an exterior object later.
- 5) Logic view: Just like "Object view", it specifically displays the current temporary logic generated by the user.
- 6) Find view: User can search the long-term memory in the brain and the result will be displayed here.
- 7) Debug view: This view can only be seen when the user pause the implementation of a logic task. It displays the debug message of the current logic.

## **Space Portal**

Space portal is basically a service program without a UI for users. But in order to display the log records, we still make a window for it.



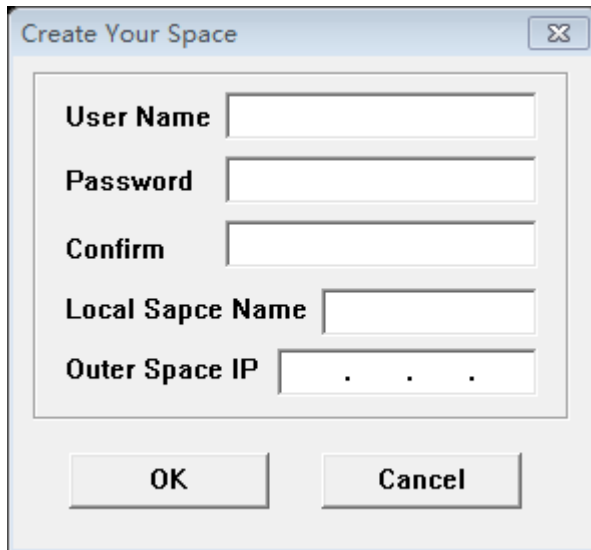
## Executer

Executer is mainly responsible for the exterior objects. When needed, the space entry program will start it to implement the tasks assigned by the space then feeds back the result to space and shut itself down automatically.

Executer doesn't need any interface but it still has a log window for debug use. It isn't shown by default; however you can see it in task manager.

## Open GMIS Instance

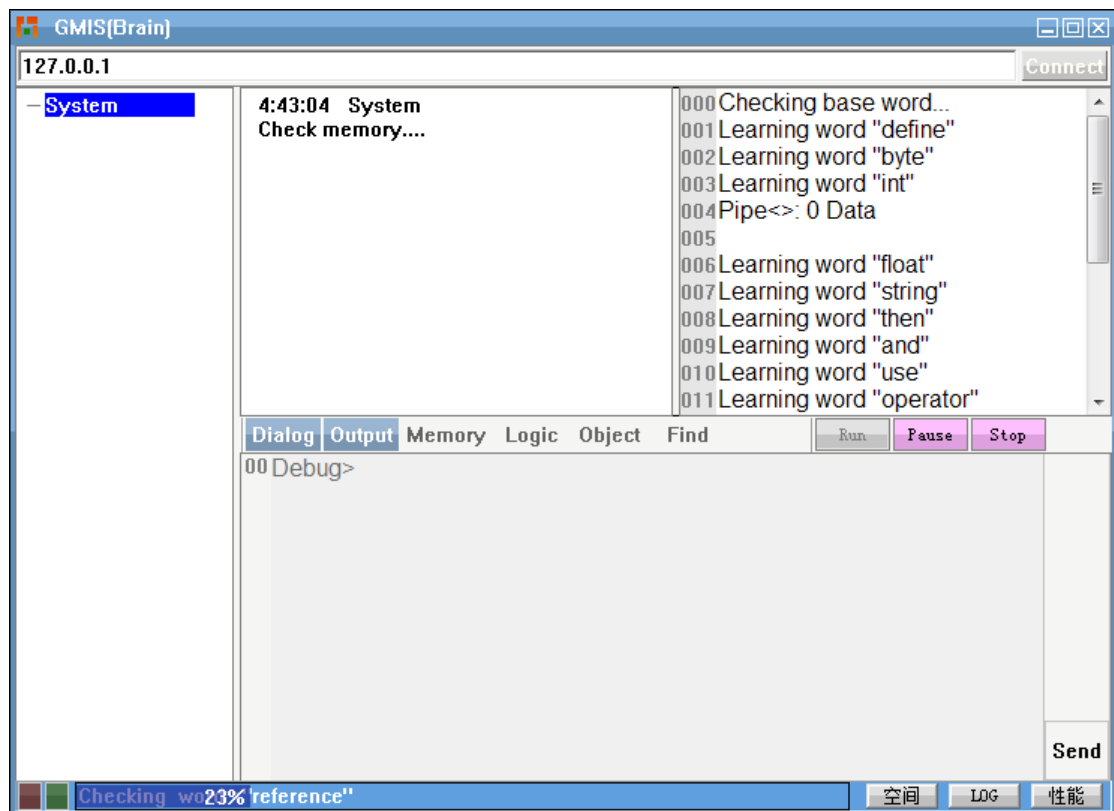
You can find GMIS.EXE program in the same folder as this manual. Start it and if it's the first time you use it, you're asked to register a user's name.



A dialog box titled "Create Your Space" with a close button (X) in the top right corner. It contains five input fields: "User Name", "Password", "Confirm", "Local Sapce Name", and "Outer Space IP". The "Outer Space IP" field has a placeholder ". . ." and a small arrow on the right. At the bottom are "OK" and "Cancel" buttons.

You can fill in "Home" or any other names you like in Local Space Name and Outer Space IP could be blank.

Then you will find that GMIS starts to check the brain's memory. As we mentioned before, besides instance, all the abilities GMIS has are acquired. It is a newborn and needs to learn some words and orders. Just be patient. Next time the start speed will be much quicker because it will only need a simple check then.



After the check, you can input your orders. But if GMIS don't connect with the digital space, it will not be able to use the exterior objects.

Click "Connect" button on top left corner to connect with the digital space. Attention that the preset IP is 127.0.0.1, which means the space entry program should be started in advance. If GMIS detected that it hasn't been running, it will be started automatically. SpaceSport.exe is in the same folder as GMIS.exe and you can start it yourself. Similarly, you need to register a user's name as the owner of the space the first time you run it.

Notice: The user's name and passport should be the same as your GMIS account because GMIS will use this name to log in the space.

When finished, you will see the address bar changed into the space name you have logged in, for example "Home". You can also click the "Space" button on left bottom then you will see a three-dimensional room:



Why it is three-dimensional? Because we hope that all the user's space can connect with each other and build a three-dimensional digital world.

Now it just a primarily project and whether it can be success won't affect the use of GMIS. Just ignore it.

Let's click "Space" button and go back to the dialog mode. Maybe you have no idea on

how to use it? Suggestion: input “define ins 23;” in input box and then click “send” button. Watch it and as the literal meaning, you have already ordered GMIS the first thing. All the orders in GMIS are combined phrase like this. Is it similar with nature language?

## Final C Language

GMIS prototype can’t understand human language directly at present, so we create Final C language to communicate with it and make it fulfill the ability of logic programming.

The first thing to point out is that users don’t need to worry about the learning of Final C and it’s only a simplified version of your mother tongue. Final C is not a new language that only designed for programmer. There’s no concept like internal storage, pointer and function and without any grammar that need to be memorized.

But as a base, some programming knowledge is inevitable. Here we assume our users have at least one high-level language programming experience.

When we were learning C language, we knew to program procedure-oriented; when we were learning C++ or JAVA language, we knew to program object-oriented; and now we propose a concept of to program logic-oriented as we are learning Final C.

We define that all logics are meaningful combination of behavior. So to program logic-oriented is actually to program with behavior units.

It’s hardly to say that behavior is totally a technological concept. There always has a meaningful action for somebody on a certain language environment while this action may just a function or a gather of functions to the computer. Not all the function or gather of functions can be called behavior.

For example, when the application implement the function of open a folder, we can say it is a behavior because we can tell another people directly:”open the folder” and he can understand it.

The difference between function and behavior is actually about granularity. At present, the granularity of programming language is function while the granularity of Final C is the function and its gather which are meaningful to human.

### Define instincts

Just like C application can be finally divided into combinations of C library function, Final C also has its own behavior unit called instinct.



We predefine dozens of instincts. One instinct may be fulfilled by one or several functions ,that's not the point ,We just need to know that these instincts belong to a machine and the meaningful basic behavior for us is the minimum set of behavior units which is necessary for GMIS to fulfill logic and keep its running.

For example “define int 23” is one of our predefined instincts. It generates an integer: 23 and without this behavior, there will be no integer for us.

Other instincts (excerpts):

```
// general orders
////////////////////////////////////
#define    INSTINCT_DEFINE_STRING          127282417796630008
...
#define    INSTINCT_USE_OBJECT              127282417797000004

// internal orders—only works within brain
////////////////////////////////////
// 1) independent internal orders--- execute singly and can't combine with other orders into a
logic
#define    INSTINCT_BRAIN_INIT              127282417798000000
#define    INSTINCT_THINK_LOGIC              127282417798000001
...
#define    INSTINCT_CLOSE_DIALOG            127282417798000008

// 2 dependent internal orders (21)
#define    INSTINCT_LEARN_TOKEN              127282417798010001
#define    INSTINCT_LEARN_ACTION            127282417798010015
#define    INSTINCT_FIND                    127282417798010019
...
```

Generally, except for some mathematical abstract behavior, these instincts is mostly internal operate for GMIS's practical use.

Now you may have a question: if all the Final C logics are combinations of instincts, could this kind of combinations fulfill the behaviors we need by robots?

For special use robots, we can customize particular instincts for it, which can not only improve the implementation efficiency, but also give diversity to robot species, just like different animals have different instincts in the nature.

As for advanced robots like “human”, they have a special instinct: can use external objects. External objects are various kinds of program modules written with high-level programming language. Just like our human beings could use all kinds of tools to achieve our goals, GMIS can inherit the technological achievements of existing IT

world by this way.

After primitive man started to use tools, they started to learn how to study, got logical ability and cooperated with each other then achieved the evolution in intelligence. GMIS robots will develop in a similar way.

As to technology, when the amount and type of instincts is determined, it is possible for the machines to programming autonomously and you are no longer need to consider about the difference between functions. All you need to care is how to use them.

## Behavior phase

Every instinct is showed by a 64 bit integer ID in brain. Obviously, it's not a pleasure work to order an instinct through an ID, which is just like to use machine directly for programming. The solution of existing programming language for this problem is to use a function name.

Using a function name to replace ID increases the programming efficiency but have no promotion on interactions between human and robots. You may have not realized it that the source program of existing computer language only could be understand when the programmer can see it. We now need: program and drive robots by speaking and hearing.

Now that instinct is a behavior that meaningful to human beings, why can't we mark it through sentences as simply as mankind natural language?

Now Final C stipulates: all behaviors (instinct is the simplest behavior) should be marked by a behavior phase.

Behavior phase must be grammatical like this:

*verb+ [adverb] + [adjective] / [noun] + [noun]*

That means for all the behavior phases, the first word must be verb and after it there could be other words or nothing. It could be an adverb and then followed by a noun. The noun could be modifying by adjective or noun.

For example, we can use "define int" to describe the instinct of define an integer. There "define" is a verb, and "int" is a noun. GMIS needs to learn these two words first and then learn their combination. After marking instinct ID with their memory ID, we can get back an instinct ID by input "define int". Then the system will demand you to input a number after the instinct ID. So if the user input "define int 2", they can get a behavior that define integer 2.

You may notice that there's no subject. On the one hand the default subject is the robot

itself, on the other hand, there needs more discoveries about language for GMIS. It's just an expedient and a preparation for the further changes after compatibility that to prescript the grammar.

Marking behaviors can help us understand it and can help in communication with others. To use it conveniently, GMIS will mark all the instincts an English version order at the first time the user initialize it. However the users can teach it the same instinct through other languages at any time.

Besides instinct, GMIS can also regard the logic of instinct combination as a behavior. Users can mark this logic with a behavior phase and then use it in the same way as use instinct. Thus the language ability and active ability of GMIS are all learned by the user, which is similar with the growing process of human.

## Logical relationship

Here the logical relationship means the combination mode of behaviors. Different from other programming languages, for the programmer's convenience, GMIS packages the logical relations into loop, condition and sequence, which decrease the programmer's pressure but the price is that even a simple code can't be understood by listening because the expression of code is quite different from our daily languages.

However Final C only has two expression modes: series or parallel, in other word either do something by order or do something at the same time.

Take the following instinct combination as example:

*Define int 2, then define int3;*

There's two clause in this sentence and each one represents an instinct of define integer. The second instinct have "then" as the first word, which means there has a logic that to implement "define int 2" first and then "define int 3". Usually we don't input "then" for convenience and the system will regard them as series sentence.

*Define int 2, and define int 3;*

The same clauses but the second one have "and" as the first word, which means these two behaviors should be executed at the same time.

It quite match up with the language intuition of human that using series or parallel to express the logical relations between behaviors. In daily life, we can fulfill most of the demands by using these two simple behavior combinations. However it's not as easy as other high-level programming languages on using these two relations to express complex logic.

We can say that Final C makes simple simpler because in daily life most sentences we say are just some combinations of behavior phrases for which even old people can use mother tongue subconsciously to program and drive robots do 90% of the things. But it also makes complex more complexer. In order to make robots learn the complicated algorithm, we need the programmer to be more professional. Fortunately, after programming, the complicated logical could be seen a single behavior and learned by robots, thus even old people can use it easily.

## **Program logic-oriented**

Program logic-oriented is to program the logic totally by behaviors. The problem is, when there are only series and parallel relations between logics, can this way fulfill all the algorithms? Now let's prove its feasibility.

First of all, let's introduce the logic execution of GMIS by one sentence and it very similar to the circuit you had learnt in physics on middle school: one data pipeline is like a wire and the behavior combinations flow through it. Every behavior is like an electronic component and either they work in parallel or in series. Every component withdraw the demands data from the pipeline and release them back (if still exist) after the task executed.

According to the theory of structured programming, we can do all the algorithms if only we fulfill the conditional loops. So we will program to let GMIS do such a thing: define a character string s="hello", and let GMIS's brain speak it (show it in Output window) consistently for 10 times. To fulfill this task, we will use condition, loop and sequence structures.

### **Common instincts**

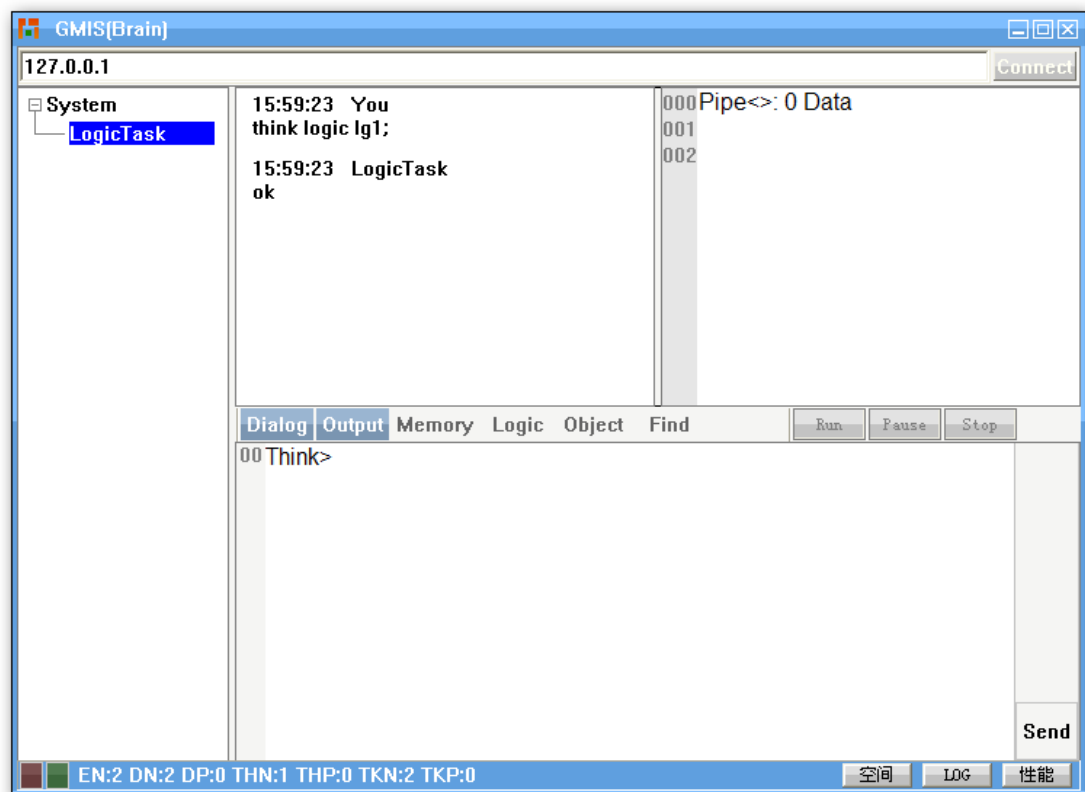
Before programming, we must introduce some instincts.

#### **1) think logic "XXX"**

This order will make the brain enter thinking cap and prepare to build a temporary logic. "XXX" is the name of the logic you want to build, by which you can quote this temporary logic in other logics. This name should be the only one in current dialog and there are no rules for format. Quotation mark is not required If the name is simple.

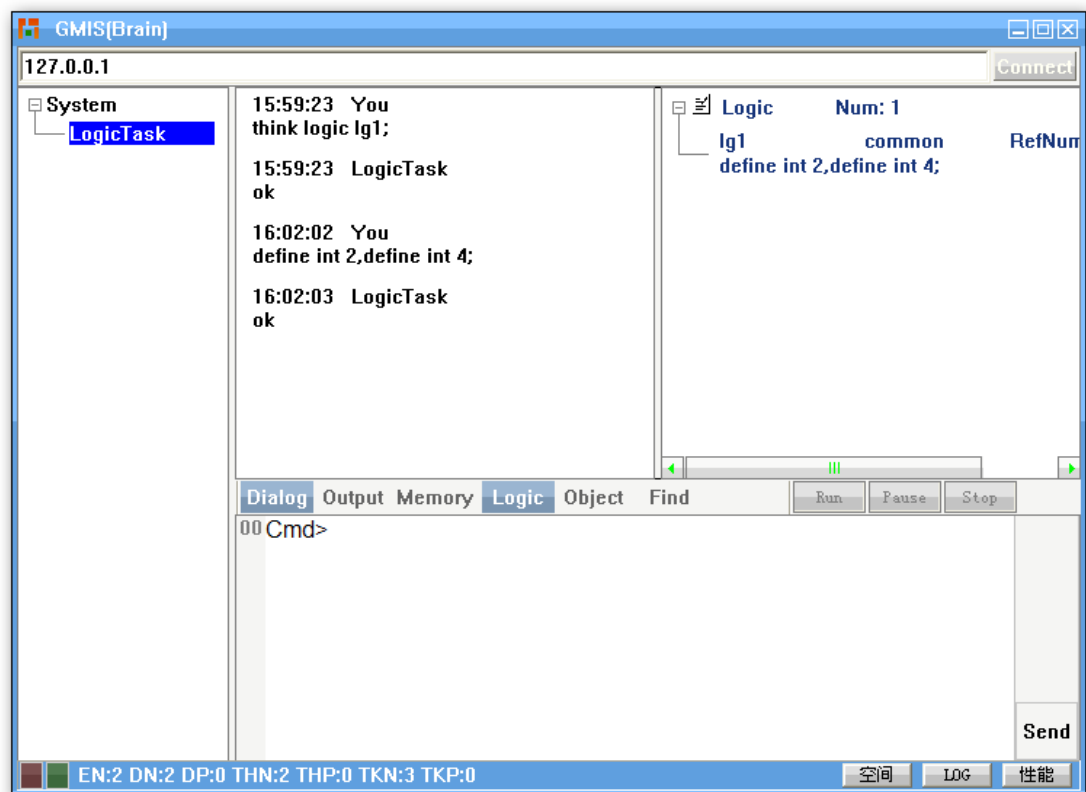
For example, we input: think logic lgl;

Then the input window will show like this:



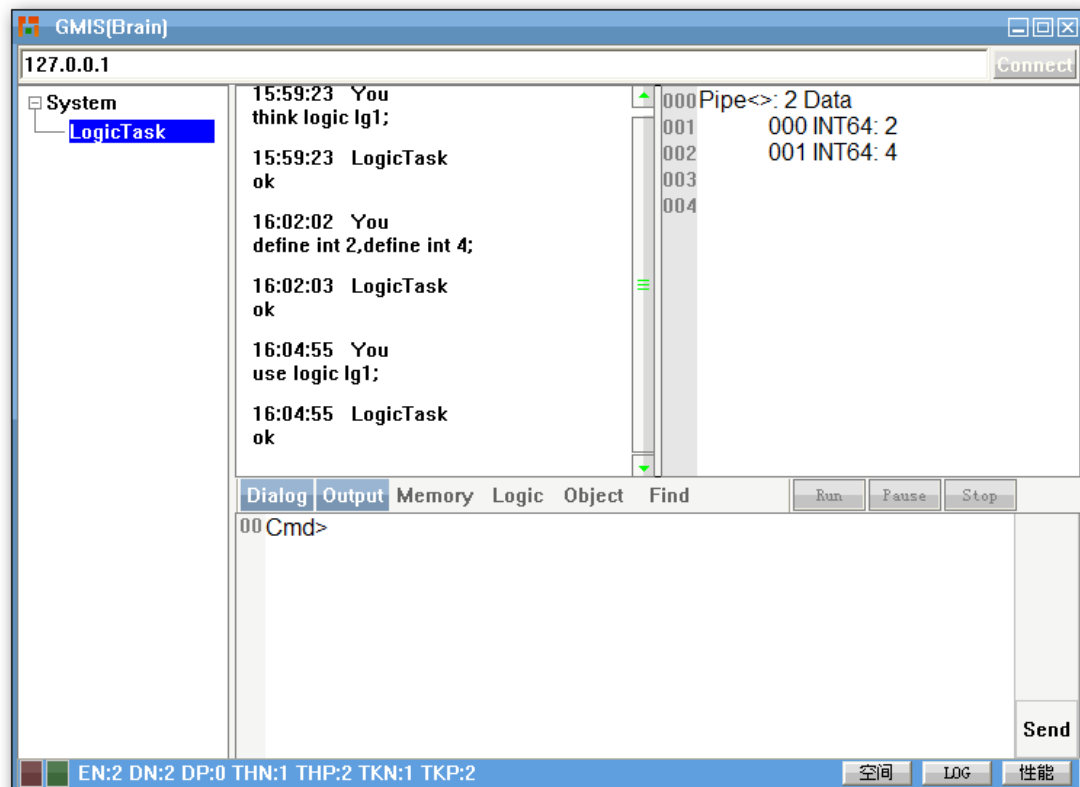
Notice: in the input window, the prompt "Cmd>" changed into "Think>".

If you input another order such as: define int 2, define int4; and after clicking "Send" you will see the temp-logic you just defined in the temporary logic window inside of implement this order instantly:



2) use logic "xxx"

The function of this order is obvious. Taking the temporary logic we just made as an example, as we input:"use logic lg1;" and implement it, you will see the result in the output window. Means that implementing: define int2, define int4;



- 3) define string "xxx"  
This order is similar to define integer and xxx is the character string which you want to define.  
Example:  
Define string "hello"
- 4) set label "xxx"  
This order means defining a label and xxx is the name of the label.
- 5) goto label "xxx"  
After using the order set label, input goto label and the system will goto the label and implement the orders after this label.
- 6) use capacitor xxx  
This order is to define a capacitor name. We all know that capacitor can: charging when the capacitance has no electricity and discharge when it has electricity. Here data pipeline replaced current and in other word, the capacitor here is actually a data TS (temporary storage) device.
- 7) Reference capacitor xxx  
This order means quoting the capacitor named xxx, that is to use the capacitor xxx again and if there's any data in this capacitor, it will be fetched.
- 8) use inductor xxx

This order is to define an inductor. The feature of inductor is: if there is a current flow through it, the inductor will be charging and the data will be saved. If there's no current, the inductor will be discharging. Here data pipeline replaced current and the inductor here is also actually a data TS (temporary storage) device.

9) reference inductor xxx;

This order means quoting the inductor named xxx and to use the inductor xxx again.

10) use resistor xxx

This order is to generate a resistor. We all know that the function of resistor is to consume current and the resistor here is to consume data. And xxx is the amount of data that to be consumed. It can be larger than the amounts that really exist. If xxx=0, nothing will happen.

11) use diode xxx

This order is to use a diode. We all know the function of diode is to open a branch selectively. Here its function is to compare the first data in the data pipeline with xxx and if they are equal, this data pipeline will be accessible. If not, this data pipeline will be shut down.

12) use operator xxx;

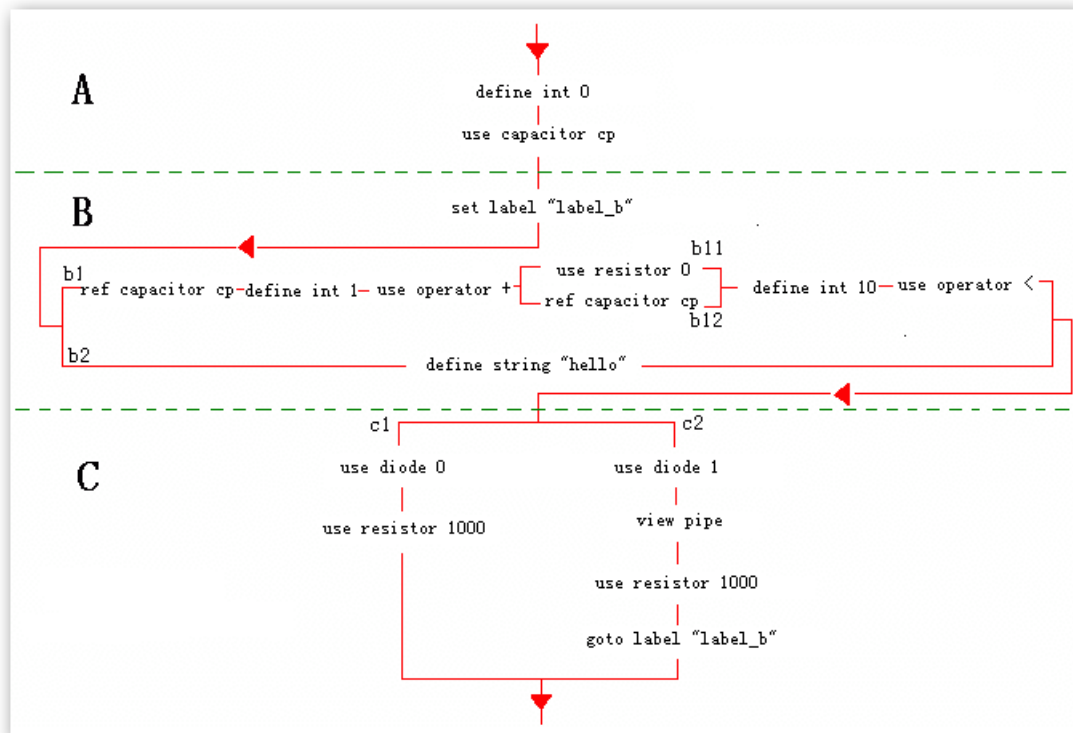
Here xxx represent operative symbols such as +, -, \*, / and so on. This order is to fulfill the corresponding operation.

### **Example for Loops**

Now let's program to fulfill this task: define a character string s="hello", and order GMIS's brain to speak it (show in Output window) for 10 times. Through this example, we can theoretically prove the feasibility that Final C can fulfill programming tasks.

There's the implementation flow chart:





We divide the task logic into A, B, C three parts.

A: define the start count of the loop. Write integer 0 into the pipeline and let it flow through capacitance CP. CP will temporary storage the data, that is to save integer 0 and the data pipeline is empty.

*think logic a;*

*define int 0, use capacitor cp;*

B: after flowing through A, the data pipeline will go through label\_b which won't have any affects on the data. Then it will go through a parallel structure b (b1, b2). Notice: parallel means the data in the pipeline will become the same two copies and flow toward b1 and b2. After implemented in b1 and b2, they will combine and flow out in one data pipeline.

*think logic b;*

*Use logic b1, and define string "hello";*

As for b1, the data pipeline first flow through the quotation of capacitor CP (actually flow through capacitor CP again). The capacitor discharges to the data pipeline and makes the defined loop count 0. The following order define int1 will define a step length, that is write integer 1 into the pipeline. Then plus these two number and get a new integer. In order to use this new integer, we quote capacitor CP to recharge and save it. This process will be realized by another nested parallel structure (b11, b12). (b11, b12) consist of a zero resistance and a quotation of capacitor CP. The data pipeline flow through this parallel structure and for the zero resistance, it will keep intact (still

include the integer just got). However the capacitor CP will be recharged when the data pipeline flows through CP's quotation and no data will flow out this branch. Thus finally only the data that flows through b11 remains. Then there's a "define int10" behavior flows out from data pipeline b1 and it writes the final count of loop times into the data pipeline. Now there're two integers, one is the loop index, and the other is the final times of loop. They will finally flow through a "use operator<" behavior. The result, 0 or 1, will become the only integer in the data pipeline.

*think logic b1;*

*reference capacitor cp, define int 1, use operator +, use logic b11, define int 10, use operator <;*

*think logic b11;*

*use resistor 0, and reference capacitor cp;*

The parallel branch b2 only defines a character string "Hello";

After the data pipeline flows out from B, the two parallel data branches will merge and there will be two data: integer (0 or 1) and character string (Hello) and then they will flow toward C;

C: it's still a parallel structure (c1, c2). Its two branches all have a unilateral diode. c1 will take the first integer from the pipeline and only if it's 0 can it be implemented, otherwise it will be invalid and stopped. And c2 is on the contrary, only if it's 1 can it be implemented.

Now the selection of program is determined on the implementation result of part B. If the first data in pipeline is integer 0, the c1 will be implemented and the loop is fulfilled. We just need to set a big resistance to clear away all the data in the pipeline (the number of resistance represents the amount of data that to be cleared and obviously 1000 is large enough). Otherwise c2 branch will be selected and as the data has flown through, only one data "Hello" remains. Output character string "hello" and then set a big resistance. Finally get back to the preset label "label\_b", continue the loop.

If you still remember the circuit knowledge learnt in middle school, the whole logic loop is not difficult to understand although it is a little complex.

The entire logic list as following:

*think logic a;*

*define int 0, use capacitor cp;*

*think logic b11;*

*use resistor 0, and reference capacitor cp;*

*think logic b1;*

*reference capacitor cp , define int 1, use operator +,use logic b11,define int 10, use operator <;*

*think logic b;*

*use logic b1, and define string "hello";*

*think logic c1;*

*use diode 1, view pipe, use resistor 1000,goto label "label\_b" ;*

*think logic c2;*

*use diode 0, use resistor 1000;*

*think logic c;*

*use logic c1, and use logic c2;*

*think logic e;*

*use logic a, set label "label\_b", use logic b, use logic c;*

You can copy the above logic directly and paste it to the input box. After click button “send”, if there’s no mistake, it can be implemented entirely at once. If can’t (something wrong during copy), you can right click the input box and select “Think” of the popup menu to check which sentence is wrong. You can also click “Analyse” to check if your sentence is divided correctly.

If the logic is correct, input: Use logic e;

After clicking “send”, you will see “hello” in the Output window for 10 times (output by view pipe order). You can replace this order by whatever orders (logics) and GMIS can do whatever you ordered it in loop then.

## High-level programming

We have proved that theoretically Final C can build all complex logics. But to realize practical use, we still need to achieve following goals:

- Can build whatever data structures

We can generate data directly by instinct and release them into the data pipeline. But sometimes we need structured data for programming. The ePipeline we designed has the ability to express any data structures and we just need to design some instincts to fulfill this function.

- Can use data logically

At present, we can only limited use certain data by naming capacitor or resistance, which is equal to use a named variable. But using data logically means we need something like array that can use certain data in selectivity or circularly.

- Can alter the logic dynamically

Different from other high-level programming language, the intelligence here not only means implement tasks according to the fixed routines, but also means the brain that can alter the preset logic according to the implement result of earlier stage or the changing of environment.

- Can use exterior objects

The intelligence of a brain reflected in how will it manage and use exterior objects. The brain itself only provides the basic instincts to build abstract logic and to keep its running.

- Can specialize and cooperate with other robots

However intelligent a robot is, its ability is limited. If robots can specialize and cooperate with each other like human beings, the world will be totally changed.

## Focus

Let's introduce the concept of focal spot before the complex logic programming.

As the interior operand of the brain becomes complicated, command statement can't express entirely the operand alone. Now we can set an operand as a focal spot for operating.

Some orders will generate default focus after implementation and we can also use focus sentence to set focus.

At present the focus has following categories:

- 1) Temporary logic
- 2) Exterior objects
- 3) Ephemeral data
- 4) Person

These focuses can be existed at the same time and form a scene. Scene has great significance on intelligence and language understanding. Scene can provide subject or object to the behavior, for example sb todo sth is equal to focus sb, focus sth, to do; the brain can get the abstract mode of the behavior from scene and memorize it. The brain can also deduce backward to get the focus object through the abstract mode in memory, which can not only avoid the verb's ambiguity, but also help achieving the initiative of intelligence.

Now simplify it and our purpose of set focus is to fulfill a certain order. Now we have the focus orders as following:

1) Object focus

```
focus object xxxx;  
focus object;
```

The first order is for static object and xxx is the name of object that is to be set focus. The second order is for dynamic object and the name of object that is to be set focus will be got from data pipeline.

2) Logic focus

```
focus logic xxx;  
focus logic;
```

Similar to object focus, the first order is for static state and the name of logic will be given directly. The second order is for dynamic state and the name will be got from data pipeline.

3) Data focus

```
focus memory xxx;  
focus memory;
```

It also has two versions like the focuses above.

Notice: the present focus will keep its type until the new focus set. And for parallel implementation, when a branch is stopped, other branches will still go on, so there's a problem that the focus may be changed as implemented in other branches.

## Debugging and executing

GMIS can implement logic while debugging but this process is quite different from other high-level programming languages'.

Generally, the debug and implement of other high-level languages is just a fault correction process for the programmer with the help of debugging tool. The application program itself can't turn into the debug mode.

But just like human beings, if it's uncertain about a task, GMIS will try step by step and turn into normal implementation if there's no problem. If there's something wrong during the normal state, it will turn into debugging state and to troubleshoot while implementation. So to speak, the debug and implement of GMIS is another implement pattern as important as normal implement.

Base on the loop logic of last chapter, we alter the loop into infinite loop that we have times to stop it. That is change the logic:

*think logic b1;*  
*reference capacitor cp , define int 1, use operator +,use logic b11,define int 10, use operator <;*

into:

*think logic b1;*  
*reference capacitor cp , define int 0, use operator +,use logic b11,define int 10, use operator <;*

**Changing the step length of the loop to 0 so that it can never bigger than 10;  
The logic of changed loop is:**

*think logic a;*  
*define int 0, use capacitor cp;*

*think logic b11;*  
*use resistor 0, and reference capacitor cp;*

*think logic b1;*  
*reference capacitor cp , define int 0, use operator +,use logic b11,define int 10, use operator <;*

*think logic b;*  
*use logic b1, and define string "hello";*

*think logic c1;*  
*use diode 1, view pipe, use resistor 1000, goto label "label\_b" ;*

*think logic c2;*  
*use diode 0, use resistor 1000;*

*think logic c;*  
*use logic c1, and use logic c2;*

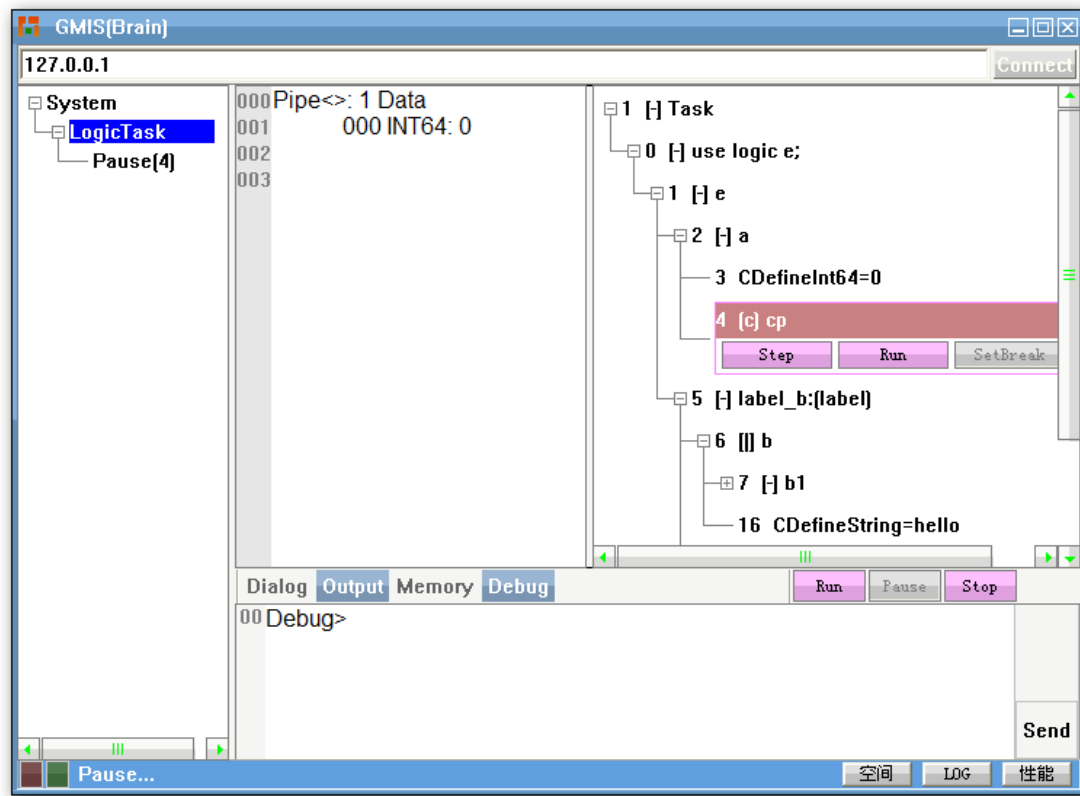
*think logic e;*  
*use logic a, set label "label\_b", use logic b, use logic c;*

**Notice: if you have already had the above logics in your current dialog, you will have 3 choices:**

- 1) Click whichever item in the temporary logic window and empty all the temporary logics by click the button in toolbar.
- 2) Click b11 item in the temporary window and delete it. Then rebuild b11 temporary logic.
- 3) Copy all logics above and paste it in the main dialog window. Main dialog will generates a new sub-dialog to implement.

Whichever one you have chosen, then input:"debug;" and the current dialog will get

into debug mode. You will find the command prompt changes into:”debug>”, and then input:”use logic e;” implement it and you will find the following debug window opening:



There's a tree in the debug window. Don't ignore this tree and it's the key for GMIS to realize automatic programming.

All logics have been assembled into a tree to implement. We know that after restructuring, a tree will still be a tree, which is as important as the neural tree to the brain for human.

After changed into debug mode, we can stop the implementation and click Step button for single step. For each step, you can see all the data of the present data pipeline in output window.

The user can also click Run button and get into normal state. As it is set infinite loop, we still can click Pause button and get into debug and implement mode.

Notice: Run button of item is a little different from which in dialog toolbar. We know that logic can implement more than two loops at the same time and click the Run button of some item can just restart the implementation of this item's loop, it has no effect on the state of other parallel branch. But if you click the Run button in dialog toolbar, the pause state of all the branches will be cancel.

Moreover, the user can also click an item and set it break, then whether it's normal state or debug implementation, it will stop implementation process at here.

## Data table

In order to build any data structure and realize logical operations of data, you need to use a data table. Data table is a kind of temporary memory and is used to save temporary data. However it's more flexible than capacitor.

We have mentioned the data pipeline for several times: ePipeline. Besides serving as implement pipeline, it's also a vessel that can save all kinds of data types. If you look over TheorySpace\Pipeline.h, you will find that it can insert integer, floating-point number, character string and other ePipeline. Thus building data structure is actually assembling of any inserted data pipeline.

There will always have an ePipeline run through the logical tree in the implementation process, providing data for every node and getting back the result data (if any). But some time we need to do some special treatment to the feedback data. We can't operate directly at current ePipeline, for it will affect the data in it. We need to transfer the data in a data sheet for operation.

The operation of data sheet involves following orders:

- 1) create table xxx;  
Generate a data sheet named xxx and it is the focus data sheet by default. This name should be the only one in current dialog and the system will check it. This sheet can be found in memory view.
- 2) import data;  
Import the data from current ePipeline to current focus data sheet.
- 3) export data;  
Export the data from current focus data sheet to current ePipeline.
- 4) insert data  
Insert the data of current ePipeline as a row into current focus data sheet.
- 5) get data  
Specifying a row of data in the focus data sheet and put it in the ePipeline. The row number is decided by the first data of the ePipeline.
- 5) modify data  
Replace specified line of data in the focus data sheet by the data from ePipeline. The row number is decided by the first data of the ePipeline.
- 6) remove data  
Delete specified line of data in the focus data sheet and the row number is decided by the first data of the ePipeline.
- 7) get size  
Return the line number of the data sheet.



- 8) close table;  
Close current focus data sheet.

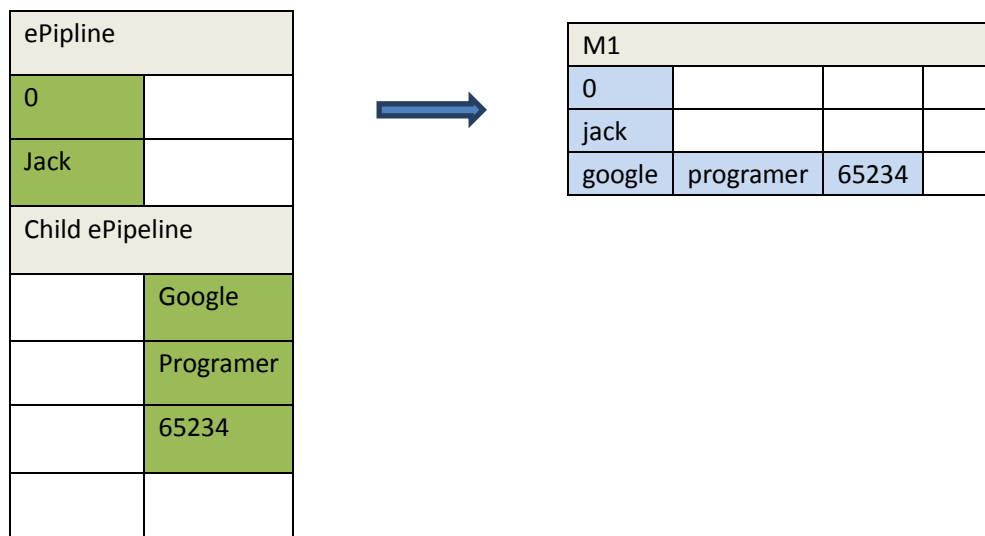
For example, if the following data is in the current execution ePipeline:

```
ePipeline{
  int32  0          //age;
  string "jack"     //name;
  ePipeline{
    string "google" //company
    string "programer" //position
    int32  65234    //salary
  }
}
```

We need to import it in a data sheet named m1 and we can input:

*Create table m1, import data;*

After execution:



As shown in the figure, every data in the execution ePipeline becomes a line in data sheet m1. If the data is the type of ePipeline and concludes several data, then the corresponding row in m1 will conclude several data. Notice that after data transfer, there will be nothing in the original data pipeline.

Accordingly, if you want to transfer the data in sheet m1 back to the current ePipeline, you can input: export data; and then the data in every row of the data sheet will be exported into the ePipeline. If there are several data in one row, they will be packaged in one ePipeline and the exported. After the export work, the data sheet will be empty.

We can use get size order to get the number of rows. Generally, we also use this order to get the amount of data of current implementing ePipeline. (notice: this amount isn't include the data in son pipeline and the answer of the above example is 3 inside of 5.)

If there have any data in current ePipeline:

```
ePipeline{
    int 4,
    int 5
    int 6
}
```

Now we want to insert the data of this ePipeline to the end of m1, then we can input:  
Use capacitor c1,Get size, reference capacitor c1,insert data;

M1 changes into:

M1			
0			
jack			
google	programer	65234	
4	5	6	

Why we need to use capacitor? Because we need the amount of rows of sheet m1 and put it to current ePipeline as the first data.

If we need get the data of a certain row, then use the following:

```
Define int 2,get data;
```

After implementation, "google""programer""google" will be pressed in orderly into the current ePipeline as three character string. Notice: it's different from export data. The latter will package the three data into one ePipeline and press it in the current data ePipeline.

Other orders are like the order get data and no more examples here.

The rest question is, how could we build a complex data? Take the data we mentioned above as an example:

```
ePipeline{
    int32  0           //age;
    string  "jack"      //name;
    ePipeline{
        string  "google"    //company
        string  "programer" //position
    }
}
```

```

        int32    65234    //salary
    }
}

```

If we have the data sheet and for building it:

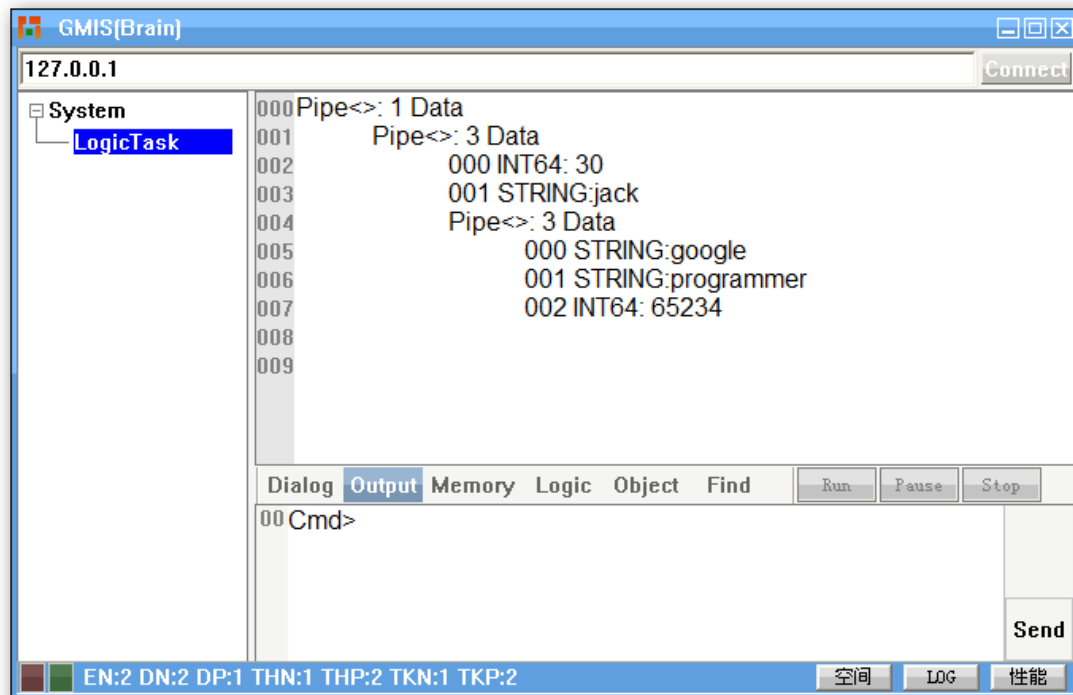
Think logic lg1;

```

create table m1,define int 30, define string jack, import data, Get size, define string
"google", define string "programmer", define int 65234,insert data, export data;

```

Implement this logic and you can see:



As a whole, the structured data could be inserted in the data sheet. Repeat this procedure and you can get any complex data structures.

## Alter logic in runtime

At the chapter debug and implement we have already mentioned that GMIS assembles all the logics into a tree for implementation. That means we can add a branch to this tree easily while implementation, or delete a branch to alter the whole logic. As I know, it is a unique feature of GMIS. With this ability, robots can change its behavior according to the change of environment.

There are two situations for altering logic in runtime. One is procedural change, which means the implementation result of the first half of logic will decide how to alter the logic in programming. For example, the logic of a manager's work is to arrange work to staff and monitoring his implementation. Once a workmate is in idle state, the manager

will arrange him task again. The manager knows that someone is sure to be in idle state but can't foreknow exactly who will. How to do with this situation? We can preset an inserted logic and if the condition is met, the preset behavior will be implemented. Procedural changes are only suitable for the situation of insert or delete logics.

The other is temporary change, of which the change demands are often from exterior instead of its logic. The logic has already been compiled and implemented and every logical element has its exclusive number as address. Therefore, it's not a difficult work to add or delete any logic branch according to this address.

Now we just discuss the first situation procedural change.

#### 1) Insert logic

Insert logic means inserting a branch into a compiled series structure and parallel structure. To do this, we should know the location of the insertion point first.

It is fulfilled by the order:

*Use logic lg1@InstancName*

We have used use logic order for several times in above examples and this is its full form:

*Use logic logicName@InstanceName:comment;*

Here logicName is a temporary logic name we generated. InstanceName is the example name of this temporary logic name, because as we use the same temporary logic in different places, we actually used its different examples. Generally we don't need this example name so it can be omitted and we can also add a comment to explain what function of this temporary logic has, and it's also optional.

For example there's a temporary logic lg1 and we can use it as this:

*Use logic "lg1@worklogic: loop logic";*

*Use logic "lg1: loop logic";*

Only after having an example name, a temporary logic can be set focus by focus logic order. And then we can do other operation toward this logic example, such as insertion.

Order: *Insert logic xxx;*

Here xxx is a temporary logic name and it is used to insert the temporary logic named xxx into the current focus logic and become its child.

There's an example:

*think logic a;*

*define int 3;*

*think logic b;*

*define string "hello world";*

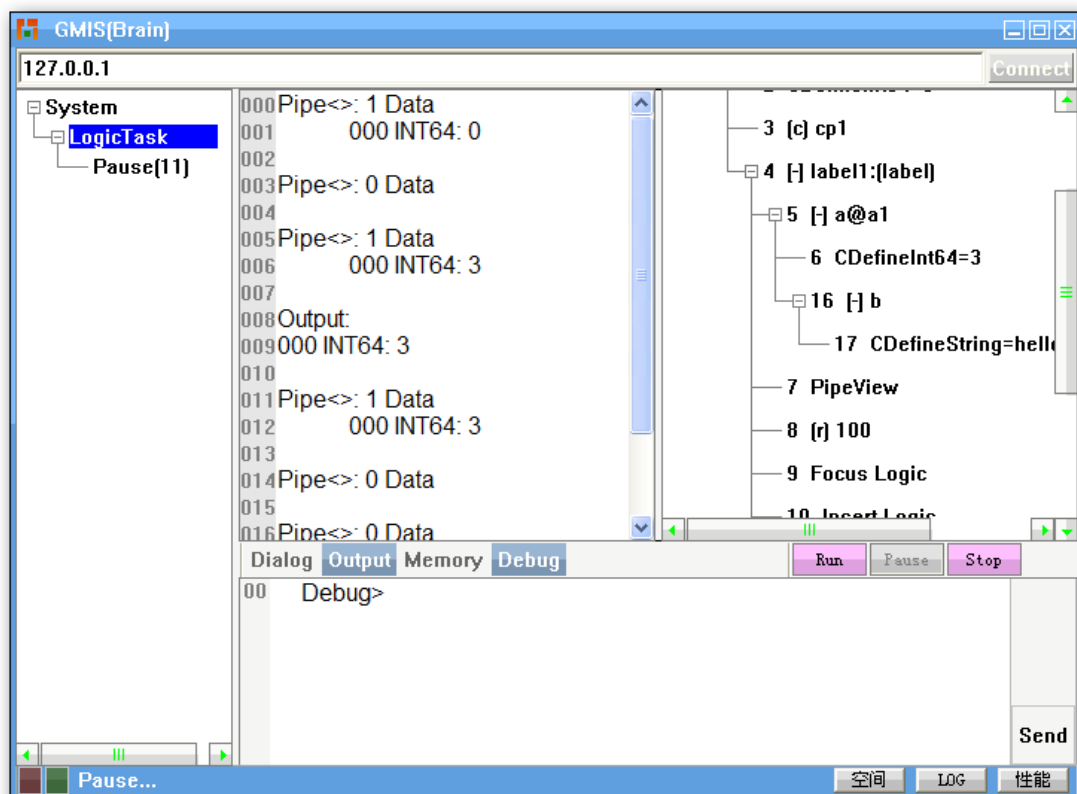
*think logic c;*

*define int 0,use capacitor cp1,set label label1,use logic a@a1, view pipe,use resistor 100,focus logic a1,insert logic b,reference capacitor cp1,use diode 0,define int 1,reference capacitor cp1,goto label label1;*

The function of this logic is to implement logic a first, then insert logic b into logic a, and again implement logic a, over. The best way to observe the implementation of a logic is to debug it.

Input “debug;”, implement, and input:”use logic c;”, implement, and you will see the debug window.

Then single step it to the item ID=10 (insert logic), observe the change of the item ID=5. After item 10 implemented, item 5 changed into:



You can see that a sub item is added to item 5 and it is logic b we inserted.

## 2) Delete logic

To delete logic we need to use this order: Remove logic;

There's no parameter for this order. It will find current focus logic in the root and remove the branch where it is in.

Let continue the above example and input the following logic:

*think logic d;*

*focus logic e1,define string "hello world",view pipe,remove logic;*

*think logic e;*

*use logic a,and use logic d;*

*think logic f;*

*define int 0,use capacitor cp1,set label label1,use logic e@e1,view pipe, use resistor 100,reference capacitor cp1,use diode 0,define int 1,reference capacitor cp1,goto label label1;*

In this example, logic e will be implemented first, after the result displayed, logic d will remove itself. When execute logic e again, the result will show that logic d is no longer existing. You can observe the change step by step in the way of debug and implement.

## Using exterior objects

Just like human can use tools to extend his abilities, GMIS also has some instincts that can help it using exterior objects----which mainly means executable codes programmed by other traditional ways, thus to realize infinite functions with the control of amount of instinct.

As time goes by, there are millions of executable codes, including all kinds of computer languages, suitable for various kinds of operating systems. It's not an easy job for GMIS to use them all.

It neither impossible for us to demand others modifying their codes to adapt to your system, nor for us to use these old code directly without any change.

The solution GMIS gives is to control the change within its ability. First we will set an actuator as mentioned for the existing codes, i.e. JAVA actuator for JAVA codes and C++ actuator for C++ code.

Then we will package this code just to give it an ability to receive parameters from ePipeline and save the implement result to ePipeline.

During implementation, GMIS will ask the space to use an exterior object and the space

will use relative actuator to use designated object according to the type of the code, and then return the result. This process is all transparent to the users and you don't have to care about how to execute. All you need to know is what object will be required.

Now let's take it as an example that implements C++ code in windows. We package the application code into a class with standard format and compile it into a DLL. This DLL is our exterior object.

#### 1) Build exterior objects

If you know about C++, you will find that packaging is an easy work. The best way to understand it is to look the source code of the example yourself. There the given code is related to other base class and we don't want to change our focus to explain it, so we just show a frame here. Don't be worried if you can't understand it, because almost nothing that we use in real life is made by ourselves and it is enough for us to use the ready-made products. Now assume that you know a bit C++.

First of all, let's design a standard C++ class. It inherits base class Mass and here what you need to realize is mainly about virtual bool Do (Energy\* E), and this virtual function is in charge of the implementation of other codes that you need to package.

```
class MyObject : public Mass {  
  
    virtual bool Do(Energy* E){  
  
        ePipeline* Pipe = (ePipeline*)E;  
  
        ...//you can fetch data from Pipe and implement whatever you want, then return the result  
        back to Pipe.  
  
        Return true; //or false  
  
    }  
}
```

Then we compile this class into a DLL and make DLL export the following 4 standard functions.

```
extern "C" _LinkDLL Mass* CreateObject(std::string Name,int64 ID=0){ return new  
MyObjec(Name,ID); }  
  
extern "C" _LinkDLL void DestroyObject(Mass* p){ delete p; p = NULL; }  
  
extern "C" _LinkDLL DLL_TYPE __stdcall GetDllType();  
  
extern "C" _LinkDLL const TCHAR* GetObjectDoc(){ static const TCHAR* Text = "This is MyObject's  
document"; return Text;}
```

Here the user can copy the first 2 functions but for the third function, you need to give the DLL type and use `GetDLLType()` to match the corresponding actuator. For example if the runtime library you connect is debug version, then it should be an actuator of debug version to implement it. It is also a standard function. What need the user to compile oneself is `GetObjectDoc()`----introduce the actual functions and usage of the packaged objects to unknown users.

Now an exterior object is almost done. Obviously, it can not only inherit the old world, but also generate new objects. Maybe some people thought that the implementation efficiency of Final C is not high enough, however the function of a brain is not mainly on efficiency but on flexible control. We can package the codes that need high efficiency into exterior objects to implement.

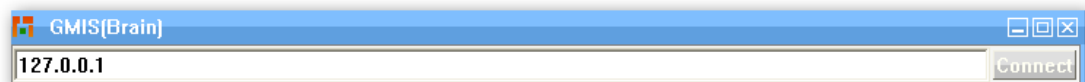
## 2) import exterior objects into digital world

Still remember the concept of Space? Because at present GMIS has no information organ and can't perceive the real world, we must set it a digital space so that it can interact with this digital space directly.

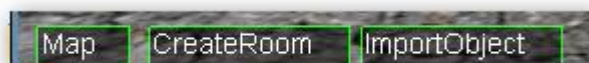
Now the digital world is a desert and we need to import the exterior objects we made.

First we need to make the DLL you packaged into a ZIP file. Although maybe most of the exterior objects are monofiles, there will be some objects have its auxiliary file, so we need to import them in the form of ZIP. Maybe in the future we should give this kind of ZIP a unique suffix, just like Android gives its App the suffix of APK. However just simplify everything now.

Then connect GMIS with the digital space and the default address is 127.0.0.1, which means the to be connected space entry program is in local.



After connected, click the “space” button in low right and you will see a three-dimensional scene. It's a pre-release and we hope that finally every one's space can interconnection and build a three-dimensional digital world.





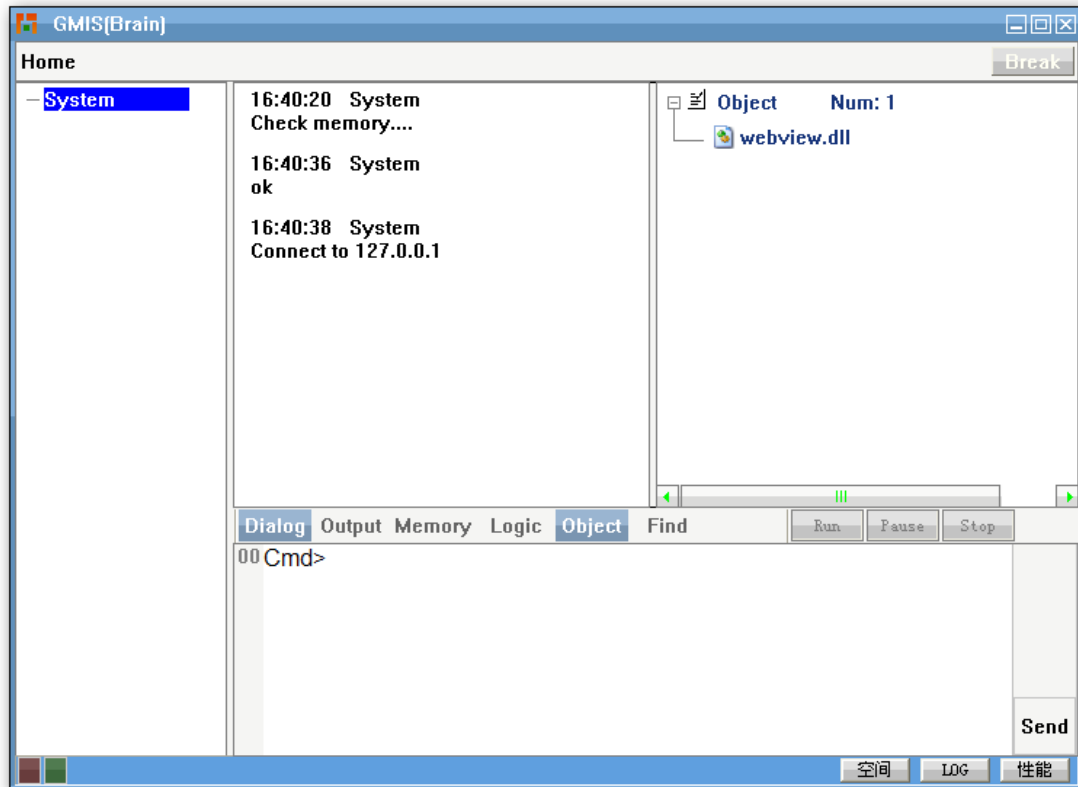
Click “ImportObject”, FileDialog pops up. Choose the appointed zip file and the exterior object will be imported into the current space. The system will generate a three-dimensional object by default.

### 1) Use exterior objects

Just like what human will do when using an exterior object, you need to find it and select it. GMIS need to do the same, however you have to help it to do these. First right click in current three-dimensional scene and there will be a ListView that shows the list of all objects in this room. Then click item and there will be a toolbar, click select and this object will be selected into the temporary memory of the brain.



As shown above, after click “Select”, go back to the dialog interface and open ObjectView you will see:



We can see this object in ObjectView and then we can use it directly. The instinct orders about using exterior objects include:

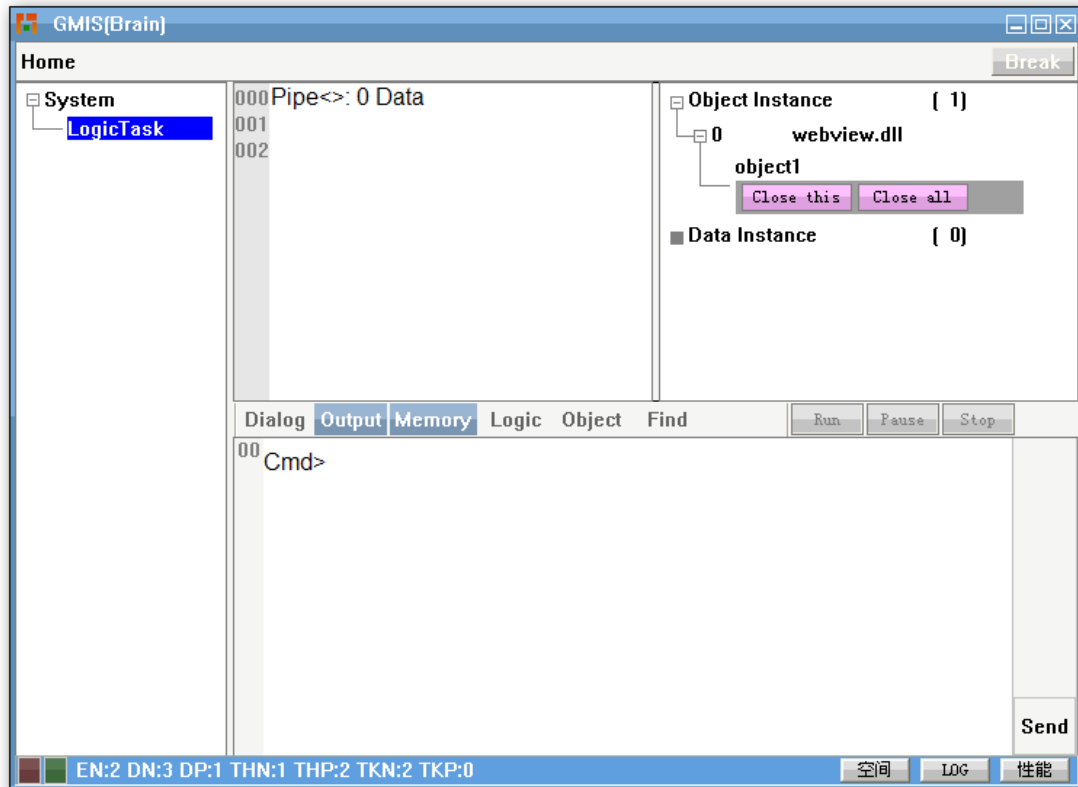
- 1) *start object "ObjectName";*
- 2) *name object "ObjectInstanceName";*
- 3) *focus object "ObjectInstanceName";*
- 4) *Get object document;*
- 5) *use object;*
- 6) *close object;*

The order start object can generate an exterior object project. Different from the objects in real world, when GMIS selects an object, it may use many projects of this object. We must name these projects for distinction purpose. After the implementation of start object, the generated object will become a default focus object, and then we can name it with the order name object.

For example input and implement:

*Start object "webview.dll",Name object "object1";*

You can see the following in memory view:



The order focus object is used to change the focus object explicitly. We have mention in the chapter of focus that a focus can be static and dynamic, for example:

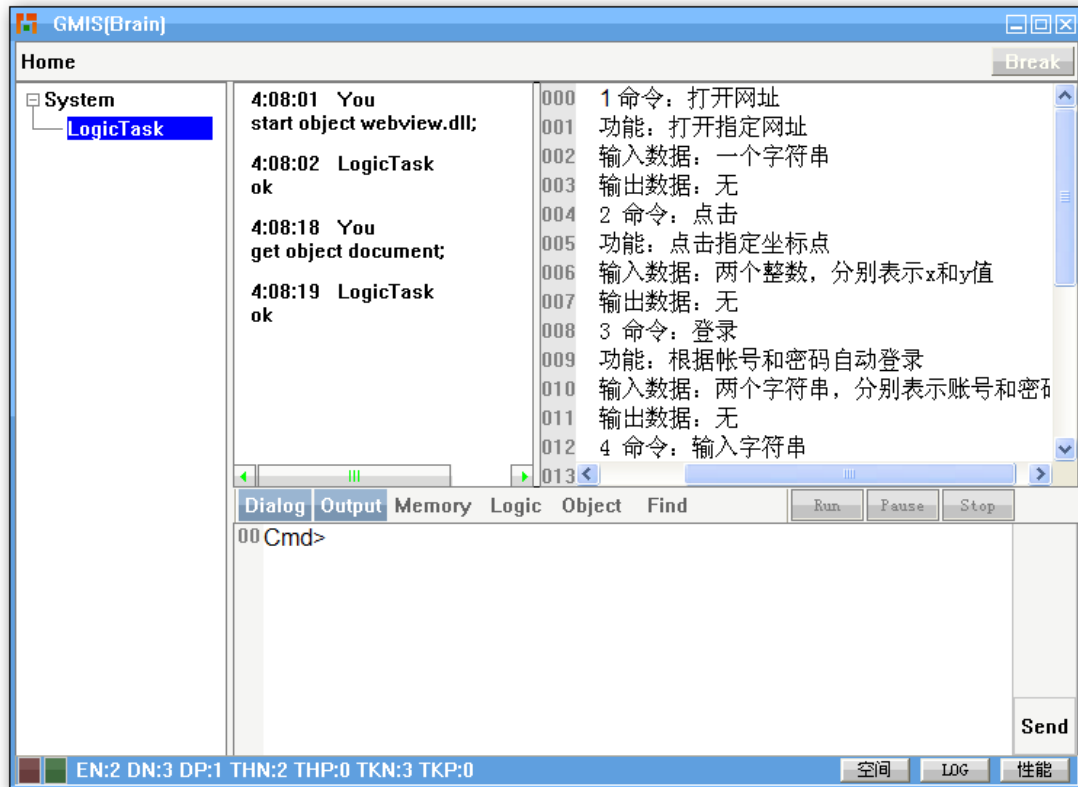
Static: *Focus object "object1";*

Dynamic: *define string "Object1", focus object;*

Actually, after we start object, the newly generated object project will be the current focus by default. Thus we can immediate use the order name object to name the current focus object. Before changed, the current focus object will be existed all the time.

Now the question is, how we can know the function of an exterior object, especially it may not generate by ourselves.

Very simple, we use the order "Get object document;" to get information:



You can see that in output window, we get all information about its function. For example we can use the first order: “打开网址” as following:

*define string “打开网址”; define string “www.baidu.com”, use object;*

Input and implement the logic above, you will see:



Actually WebView.dll simply packaging the WebBrowser controls.

Now you can use the order Use object to assemble orders the documents makes. Because it is the actuator that implements exterior objects, no matter how outrageous the logic is, the brain of GMIS won't be hurt by it.

There still a lot to do with the details on using exterior objects and at present we just want to prove that GMIS can extend its functions by this way. Now we need to package more objects with the existing open source applications in the internet, so that the people who don't know C++ can also use Final C directly to program and get the result he wants.

## Cooperate with other robots

The principle of cooperating with other robots is similar to using exterior objects. Just regarding exterior robot as an object and send orders to it. It will work as an ordinary exterior object, implements and returns the result.

The difference is that robot may check your orders and decide whether to cooperate with you. Because this need multimachine's Joint Test and we remain it to the next version.

## Application example

### ➤ Clicking a website timely

Now we will use the object webview.dll to display how to realize clicking a link or button of a website at a fixed time and you can extend it with other tasks, such as seckilling, timing to send tweet or topstick and so on.

For new instinct orders, you can view the instinct list in appendix as reference. If you don't know the function of webview.dll, take the order get object document as reference.

The logic is given directly as following:

*think logic lg1;*

*start object webview,define string "view",name object,define string "打开网址",define string "www.google.com",use object;*

*think logic lg2;*

*input num " Please enter the X coordinate:",input num " Please enter the y coordinate:",use capacitor point,input text "Please enter time: (like 23:00:00)",use capacitor time,set label label1,use resistor 10,reference capacitor time,use resistor 0,and reference capacitor time,get time,use operator ==, use logic lg3,and use logic lg4;*

*think logic lg3;*

*use diode 1,define string "刷新",use object,define string "点击", reference capacitor point,use object;*

*think logic lg4;*

*use diode 0,wait second "0.1",goto label label1;*

Finally : *use logic lg1,use logic lg2;*

## ➤ **Playing cards automatically**

In order to display the logical control ability of GMIS more entirely, we make GMIS play Texas Poker on web automatically. Of course we must package this game into an object so that to accept the control of GMIS. Meanwhile the tessseract ocr library that open sourced by google is used to recognize cards.

Our task is to make GMIS control multi-account and log in more than two accounts, enter the selected room and plays card automatically. Each account will quit the game after played specified rounds, until all the accounts fulfilled the task.

We can also make these accounts send their hand pattern to GMIS and GMIS can decide how to play a hand by unified command so that improve win rate significantly. For it requires more work, now we just simply show GMIS's ability of logical control. If replacing the object of playing cards into other robots or objects, GMIS can control these objects and robots to fulfill specified task by cooperation.

### ***Traditional programming mode***

Because it a complex logical task, we'd better simply design it with other high-level programming code of language first and then translate it into Final C logic.

```
vector<Player> PlayerList;
vector<User>  UserList;

int  PlayerIndex = 0;
bool bHasSite = true;
int  RoomID = 100;
int  PlayNum = 100;
int  CurPlayNum = 0;

Start thread 1.do();
thread1::do()
{
    while(1){
        if(bHasSite){
            for(int PlayerIndex=0; i<PlayerList.size(); PlayerIndex++){
                Poker = PlayerList[PlayerIndex];
                if(Poker.m_IsIdle){
                    if(UserList.size()){
                        User = UserList.pop();
                    }
                }
            }
        }
    }
}
```

```

        start thread 2.do()
        }else{
            return;
        }
        break;
    }
}
}else{
    sleep(100);
}
}
}
thread 2::do()
{
    Poker = PlayerList[PlayerIndex];
    Poker.m_IsIdle = false;
    if(!Poker.Login(User)){
        Poker.m_IsIdle = false;
        return;
    }

    if(!Poker.goto(RoomID)){
        Poker.m_IsIdle = false;
        return;
    };

    if(!Poker.Sitedown()){
        Poker.m_IsIdle = false;
        return;
    };

    for(int CurPlayNum=0; i<PlayNum; CurPlayNum++){
        Poker.PlayOnce();
    }
    Poker.m_IsIdle = true
    return ;
}

```

It's not complex to use traditional programming to fulfill this task. There's mainly two data structure: user's account and real players and they form into two arrays and some temporary variable minor control it. Then the two thread functions will coordinate cooperation, one in charges of assembling tasks, the other in charges of executing tasks.

### **Final C pattern**



There are two experiences for Final C to translate the above code.

First experience: using subdivision to build logic. Consider how many parts the task should be divided into and each part has sub-logics. And use the same subdivision process to each sub-logic until all the logics fulfilled.

For example: divide this playing card task into three parts. The first part prepares data, the second part prepares object project for playing card and the third part will control the playing process. Then the first logic can be like this:

*Think logic "lg1: start work";*

*Use logic "lg2: prepare account", use logic "lg3:prepare object project for playing card", use logic "lg4:work logic";*

Here logic lg2, lg3, lg4 haven't been fulfilled yet, but we can make a statement first. We will do the same work to fulfill lg2 and in this way, the whole logic will be finished fluently without jumping between header files like traditional high-level programming languages.

Second experience: mark serial number for the logic name. With the number we can roughly judge the correlation between sub-logics. The closer to the root logic, the bigger the logical series number is.

For example, we can rewrite the above logic like this:

*Think logic "lg1: start work";*

*Use logic "lg100: prepare account", use logic "lg200: prepare object instance for playing card", use logic "lg300: work logic";*

If the first sub-logic lg100 concludes its sub-logic, then we can name its sub-logic with increasing number like lg101 and we could know that the sub-logic with a series number between 100 and 200 is relative with the task of preparing account. Surely if there are too many logical levels, this way of naming will be also helpless. We have to remind here: Final C is not a new programming language that aims at prompt the work efficiency of programmer. Our ultimate goal is to realize the robot's automatic programming. We can use a unique ID to replace the naming inner brain, however we just have to do this work manually at present stage.

The whole logic code as following:

Think logic lg1;

Use logic "lg100: prepare account", use logic "lg200: prepare object instance for playing card",  
use logic lg300@worklogic, use resistor 10, define string "game over", output info;

Think logic lg100;

create memory UserAccountList,

get size, define string "account1", define string "password1", insert data,

get size, define string "account2", define string "password2", insert data,

get size, define string "account3", define string "password3", insert data,

get size, define string "account4", define string "password4", insert data,

get size, define string "account5", define string "password5",

insert data;

Think logic "lg200:create five object instance";

start object poker, name object player1,

start object poker, name object player2,

create table ObjectList,

get size, define string "player1", insert data,

get size, define string "player2", insert data,

Create table RunFlagList,

get size, define int 0, insert data,

get size, define int 0, insert data;

Think logic lg300;

use logic "lg301:control logic", and use resistor 0;

Think logic lg301;

define string "0", use cp RoomID,

use cp CurRoomID,

define string "0", use inductor ActualRoomID,

define int 1, use inductor bHasSeat,

use cp UserAccount,

define int 1, use cp bAlive,

define int -1, use cp ObjectIndex,

use logic "lg302:loop";

Think logic lg302;

Set label b1, use resistor 100, reference cp bAlive, reference cp bAlive, and use resistor 0, use diode 1,

use logic "lg303:check blank seat", goto label b1;

Think logic lg303;

use resistor 10, Reference inductor bHasSeat, Use logic "lg304:unavailable seat", and use logic

*"lg305: available seat";*

*Think logic lg304;*

*Use diode 0,define int 0,reference inductor bHasSeat,wait second 2;*

*Think logic lg305;*

*Use diode 1,define int 0,reference inductor bHasSeat,reference cp ObjectIndex,use resistor 100, define int -1,reference cp ObjectIndex,Use logic "lg306:find available seat";*

*Think logic lg306;*

*set label b306:, reference cp ObjectIndex,define int 1,use operator +, use resistor 0, and reference capacitor ObjectIndex, use logic lg309, use operator <, use logic lg310;*

*Think logic lg309;*

*focus table ObjectList,Get size ;*

*Think logic lg310;*

*Use logic "lg311:loop end",and use logic lg312;*

*Think logic lg311;*

*use diode 0, wait second 1;*

*Think logic lg312;*

*use diode 1, use logic "lg313:check available object";*

*think logic lg313;*

*focus table RunFlagList,reference cp ObjectIndex,use resistor 0,and reference cp ObjectIndex,get data,use logic lg314;*

*think logic lg314;*

*use logic lg315,and use logic lg316;*

*think logic lg315;*

*use diode 1, use resistor 100,goto label b306;*

*think logic lg316;*

*use diode 0, focus table RunFlagList,reference cp ObjectIndex,use resistor 0,and reference cp ObjectIndex,define int 1,modify data, focus table UserAccountList, Get size,define int 0,use operator >, Use logic "lg317:no new account",and use logic "lg318:new account available";*

*Think logic lg317;*

*Use diode 0, reference cp bAlive,use resistor 10,define int 0,reference cp bAlive;*

Think logic lg318;

Use diode 1,use resistor 10,define int 0, get data,reference cp UserAccount ,define int 0, remove data,define string "worklogic",focus logic ,insert logic lg400;

Think logic "lg400:auto play game";

use resistor 100,reference cp UserAccount,use cp MyUserAccount,focus table ObjectList, reference cp ObjectIndex,use resistor 0, and reference cp ObjectIndex,use capacitor MyIndex,and use resistor 0,get data,use capacitor MyObjectName,use resistor 10, use logic "lg401:open game url", use resistor 10,use logic "lg402:login", use resistor 100,use logic "lg404:go in specify room",use resistor 10, use logic "lg403:get instance name",focus object,define string "SearchSeat",use object,reference inductor bHasSeat,use logic "lg405:loop game";

think logic lg401;

use resistor 10,use logic lg403, focus object, define string "OpenUrl", define string "http://apps.renren.com/boyaa\_texas/index.php?origin=103", use object ;

think logic lg402;

use resistor 10,use logic lg403, focus object, define string "login", reference cp MyUserAccount, use object,use logic "lg500:check to see if be in hall" ;

think logic lg403 ;

reference cp MyObjectName,reference cp MyObjectName,and use resistor 0;

Think logic lg500;

define int -1,use cp cp500, set label b500, reference capacitor cp500,define int 1,use operator +,use resistor 0,and reference capacitor cp500,define int 10,use operator <,use logic lg501,and use logic lg502;

Think logic lg501;

use diode 0,define string "SearchLabel", define string "大厅", define float "90.0",use object, use diode 0,input num " You are not logged in the hall, please artificial log in, input any number to end ",use resistor 10;

think logic lg502;

use diode 1,define string " SearchLabel ", define string "大厅", define float "90.0",use object, use logic" lg503:not be in hall",and use logic "lg504:be in hall";

think logic lg503;

use diode 0,use resistor 10,wait second 2, goto label b500;

think logic "lg504:refresh";

use diode 1,use resistor 10,define string "completed login",output info, use logic lg403, wait

second 5;

think logic lg404;

use logic "lg610:define room ID", use resistor 10,use logic "lg620:go into room", use logic "lg630:verify room ID", use logic "lg640:check to see if be in specify room";

think logic lg610;

use resistor 10,reference cp RoomID,reference cp RoomID,and use resistor 0,define string "0",use operator ==,use logic lg611, and use logic lg612;

think logic lg611;

use diode 1,reference cp RoomID,use resistor 1,input text "Please input room ID", reference cp RoomID,and reference cp CurRoomID;

think logic lg612;

use diode 0,reference cp RoomID,reference cp RoomID,and reference cp CurRoomID;

think logic "lg620:auto into specify room";

use logic lg403, focus object,define string "ClickLabel",define string "游戏场", use object,wait second 1,use resistor 10,

use logic lg403, focus object,define string "ClickLabel",define string "初级场", use object,wait second 1,use resistor 10,

use logic lg403, focus object,define string "ClickLabel",define string "牌场输入框", use object,wait second 1,use resistor 10,

use logic lg403, focus object,define string "InputText",reference cp CurRoomID,use object,wait second 1,use resistor 10,

use logic lg403, focus object,define string "ClickLabel", define string "牌场搜索", use object,wait second 2,use resistor 10,

use logic lg403, focus object,define string "DbClickLabel",define string "牌场选择", use object, wait second 5,use resistor 10;

think logic lg630;

use resistor 10,use logic lg631,and use logic lg632,use operator "==";

think logic lg631;

reference cp RoomID,reference cp RoomID,and use resistor 0;

think logic lg632;

use logic lg403, focus object,define string "GetRoom ID",use object;

think logic lg640;

use logic lg641,and use logic lg642;

think logic lg641;

*use diode 1,define string "has been entered room",output info;*

*think logic lg642;*

*use diode 0,use logic lg643,and use logic lg632,use operator "=",use logic lg644,and use logic lg641;*

*think logic lg643;*

*reference inductor ActualRoomID,reference inductor ActualRoomID,and use resistor 0;*

*think logic lg644;*

*use diode 0,input num " Please manually select the room input any number to end ",use resistor 10,use logic lg632,reference inductor ActualRoomID,use resistor 10;*

*think logic "lg405:loop game";*

*Define int -1, use capacitor Index, set label b405, reference capacitor Index,define int 1,use operator +, use resistor 0, and reference capacitor Index, define int 10, use operator <, Use logic lg406,and use logic lg407;*

*Think logic "lg406:game end";*

*use diode 0, use resistor 100, define string "game over",output info,use logic lg403, focus object,define string "ClickLabel",define string "站起", use object,use logic lg403, focus object,define string "ClickLabel",define string "大厅", use object,wait second "2.8",focus table RunFlagList,reference cp MyIndex,define int 0,modify data, use logic lg409, focus logic worklogic,remove logic;*

*Think logic lg407;*

*use diode 1, use logic "lg408:play one round", goto label b405;*

*Think logic lg408;*

*use resistor 100,use logic lg403, focus object, define string "PlayRound", use object, output info, use logic lg403,focus object,define string "SearchSeat",use object,reference inductor bHasSeat;*

*think logic "lg409:check seat";*

*set label b409,use resistor 10,reference inductor bHasSeat,reference inductor bHasSeat,and use resistor 0,use logic lg410,and use logic lg411,use resistor 10;*

*think logic lg410;*

*use diode 0,use logic lg403,focus object,define string "HasSeat",use object,reference inductor bHasSeat, and use resistor 0,use diode 0,wait second 3,goto label b409;*

*think logic lg411;*

*use diode 1,use resistor 10;*

### **Notice1:**

Due to the website has recently made some changes to the game, this logical task can't be implemented as before (requires verification code and can't login automatically). There's nothing wrong with GMIS and the logical code here is just an example for your reference.

### **Notice2:**

Replace "account \*" and "password \*" in the logic code by your own account before use it after selecting "poker" as current object;

Execute: *use logic lg1;*

## **Memorizing and learning**

You will see an initialization process when first open the GMIS prototype. Because at the beginning, all instincts exist in the form of ID, we must mark it with at least one natural language so that the user could be able to use it. And this involves the memorizing and learning of the brain.

Memorizing and learning is actually a huge issue, the current solution hasn't reflects the newest research results, nor reaches the ultimate goal, however it has some practicability.

Let's introduce the current available parts.

### ***Memorizing words***

If a token will be used to build new orders, GMIS must learn its part of speech first, thus to go through the syntax check. Obviously, the knowledge of speech is too advanced for an early stage robot, but in order to standardize user's usage habits and avoid repetitive learning for a trained robot, we have to imbed it.

Order: *Learn [speech] "token";*

For the "token" you need to learn, you need to input its part of speech in the place of speech, for example Learn verb "set"; or Learn noun "time".

The available speech includes: token, pronoun, adjective, numeral, verb, adverb, article, preposition, conjunction, interjection.

Actually, we can only real use verb and noun, and token is also a speech that used to memorize specific symbols.

Example:

*learn verb "open";*

*learn noun file;*

### ***Memorizing text***

Order:

1) *Learn text "string";*

2) *Learn text;*

There are two versions. The first version is to memorize directly the character string in quotation marks. The second version is to generate a sub-dialog that demands you input the text that to be memorized.

Here the text will be divided into single characters and be memorized according to the logic level divided by punctuation marks.

Example:

*Learn text "hello world"*

### ***Memorizing logic***

Order: *Learn logic "xxx";*

Here xxx represents a temporary logic name. We have talked about how to generate a temporary logic and with the logic name, GMIS can save the specified temporary logic into a long-term memory.

For further retrievable use, the user will be demanded to input texts to describe this logic and GMIS will correlate this text memory with logic memory.

In fact, this instinct needs more researches for there're many situations for temporary logics. Some temporary logic is not independent at all and it's relay on other logics or exterior objects. Then there are more problems to be solved in memorizing and retrieving.

### ***Learn new behavior***

Now we can program with Final C to get various kinds of temporary logics, and we can also memorize these logics into long-term logics. However to use a long-term logic, we need to search its description and select the right one in the search results. Obviously



it's not convenient enough.

And for some behavior logic, it already represents a meaningful behavior. If we want to use it by orders just like use instincts, we need to learn it first.

Surely, you should program this temporary logic first and name this logic a behavior sentence. For example if you have logic of opening documents, then you may want to use it like this: open file. And you need to memorize the verb open and the noun file, finally correlate the memory of "open file" with the memory of temporary logic.

Order: *Learn action "xxx";*

Here xxx represents a temporary logic name. GMIS will memorize the specified temporary logic into a long-term memory and then require you to input the new order such as "open file". GMIS will verify whether it confirms to the grammar of predicate + object and memorize it if there's no problem. Then GMIS will correlate this order with the temporary logic or order's ID that memorized before.

Next, you can just input "open file" to execute a complex logic. It's a significant function that can help users to train and improve GMIS's ability, which makes GMIS have different abilities although they may have the same instincts. There's a big difference with traditional concepts which means almost sameness of copies provided by the programmer.

There's a pity that the realization of this order needs to be improved and the main problem is about the dependence on exterior objects. If it relies on exterior objects, the order of learning won't be executed in a simple way. Of course, many exclusive use robots don't involves the using of exterior objects and this order still has its practical meanings.

### ***Memorize objects***

We can make the brain memorize an exterior object, including its location and description, thus we can use it directly when needed by selecting the memory.

Use the order: *Learn object "objectName";*

Just like the process of learning logic, after executes this order, the user will be demanded to input a text to describe this object and then the brain will correlate the text memory with the object saved before.

If this object is a DLL generated by the above rule, then the system will autocall the built-in document of this Object to the current edit area, thus the user can edit it on these built-in texts as the description of this object.

Notice: at present the meaning of this order is not obvious and the users had better ignore it. The memory pattern of GMIS still need to perfect and the current realization is just a tentative try.

### ***Learning memories***

Learning memories means to learn an existing memory with another memory, and this ability is essential for an intelligent agent. Information has several patterns of manifestation, for instance the information received by vision sense and auditory sense are quite different although they may point to the same object, for which we need to correlate the two information. Another instance is that when we learn a foreign language, we can match words of the new language with words with corresponding meanings in our mother tongue.

Order: *Learn word memory "xxx";*

Here xxx represent the existing memory and now it could only be a word, however in the future it can be an order or a description. The system will analyze automatically whether xxx is a word, and then ask the user input new description. The new description should be equivalent in content with the old description, for example if the old memory is a word, then the new description should be a word too.

With this function, the users can use his mother tongue to replace the English orders that are defaulted by the system. For example, let's show how to use Chinese “定义 整数” to learn “define int”. we know here “定义” is “define”, “整数” is “int”, and we can execute the following orders successively:

*Learn word memory “define”, and then input “定义”.*

*Learn word memory “int”, and then input “整数”.*

Now it can be executed if we input “定义 整数 2”. We can even input like this: “define int2, 定义 整数 3, define 整数 4;” and it can be executed exactly too.

Notice, for languages like Chinese that can't be divided by spacing, users must input spaces by oneself.

## **Retrieving memories**

Because the memory pattern of GMIS still needs to perfect and like the saving of memories, it's possible for memories retrieving to have any changes in the future.

As for the current designed pattern, memories' retrieving has already give us many

surprises and it can select memories by controlling the granularity of memories so that to accelerate the retrieving speed.

For example, the granularity of red Ferrari is obviously different from which of red pencil in brain. If regardless of this and just use red as keyword for search, then maybe the result list will be too long for us. But if we roughly set a granularity and the memories beyond it will be excluded and then the item we want can be selected more quickly. That makes the brain more like a camera with adjustable focal length which can quick positioning the area where the target in. This function alone can bring a huge value in the improvement of the existing search engine. We hope further development on this basis. At present, we just provide the following basic orders:

```
find "text";  
find logic "text";  
find object "text";
```

Here “text” represents the keyword description of what you want to search. The word or and not control the search algorithm, which is similar to the usage of ordinary search engine. i.e.: find “aaa bbb, or ddd ccc, not eee fff”;

GMIS will display the eligible long-term memories in Find window of the dialog. Obviously the last two orders are used specially to search logics and exterior objects. To control the granularity of the searching information, you can use this order before search: find precision xxx;

Here xxx should be replaced by the size of searching information and its unit is second, default=5. Notice: the following searching order will be affected by this setting, unless you set a new one.

Beside we can also set the time horizon for searching memories and it can improve on the accuracy of the searching results. For workload consider, the current version hasn’t realized this.

Example:  
*Find "define";*

## Logic retrieve

Logic retrieve means GMIS can continue the execution of logic after the task is stopped during executing process, even the system and computer is shut down. During this time, the task will be saved as static data and won’t occupy other system resources.

This ability is essential to some key tasks, for example some remote computer need to

upgrade the system or restart the computer, however it don't want to restart and redo the current task, then the logic retrieve ability of GMIS can simply solve this problem.

On a long view, it's also a necessary ability in intellectual development. Everyone thinks many logics in everyday and most of which just have a beginning without any execution for various kinds of reasons. But our human beings can save these to be continued tasks into static memories and don't bring any burdens to the brain. Someday as the conditions satisfied, these tasks will be continued, or just as a reference for other tasks.

Let's simply test this ability. First input:

*debug;*

execute it and enter the debug mode, then input:

*Define int2,define int3,define int4;*

It will stop at the second behavior after executed. Now we shut down GMIS and if you want you can shut down your computer. Restarts GMIS and you will find that task dialog still exist. Select it as focus and you will see the debugging tree, you could either do step by step executing debug or click run button. The task will continue to execute from where it stopped.

For GMIS, logic retrieve is actually a memorizing and retrieving behavior. Now we just prove that GMIS has this ability and if making a good use of this ability, it can fulfill more tasks. Unfortunately, there're still many work to do to make GMIS a practically use value. For instance, if using exterior objects, just retrieving execute inner the brain is not enough. But for exclusive robots, if all behaviors are instincts, this ability is still workable

# Appendix

## Learnt words

Here we list the words (and speech) that was learnt by default at the first time GMIS prototype starts. There're only English and corresponding Chinese and the user can teach GMIS more words, add other part of speech to existing words, or translate the words in this list into your mother tongue words and send them to GMIS prototype so that you can programming in your mother tongue to drive GMIS.

Speech	English	Chinese
noun	int	整数
	float	小数
	string	字符串
	operator	操作符
	resistor	电阻
	rs	电阻
	inductor	电感
	id	电感
	capacitor	电容
	cp	电容
	diode	二极管
	dd	二极管
	label	标签
	pipe	管道
	text	文本
	num	数字
	second	秒
	table	数据表
	data	数据
	size	大小
	logic	逻辑
	date	日期
	object	物体
	dialog	对话
	token	符号
	pronoun	代词

	adjective	形容词
	numeral	数词
	verb	动词
	adverb	副词
	article	冠词
	preposition	介词
	conjunction	连词
	interjection	感叹词
	noun	名词
	text	文本
	action	行为
	time	时间
	end	终点
	precision	精度
	info	信息
	document	文档
verb	define	定义
	use	使用
	reference	引用
	wait	等待
	set	设置
	goto	转向
	view	查看
	input	输入
	create	制造
	import	导入
	export	导出
	focus	关注
	insert	插入
	modify	修改
	get	得到
	remove	移除
	close	关闭
	name	名字
	start	启动
	think	思考
	learn	学习
	run	运行
	debug	调试
	stop	停止
	pause	暂停
	step	单步

	find	搜索
	output	输出
Logic realation	then	然后
	and	同时

## 本能列表

Type	Command	Function	Input\Output	Remark
General command	Define int xxx;	put the defined integer xxx into execution pipe	Input: none Output: integer	
	Define float xxx;	put the defined float xxx into execution pipe	Input: none Output: float	Use quotation mark for decimals.
	define string xxx	put the defined string xxx into execution pipe	Input: none Output: string	
	Use operator xxx	xxx is the operator and it could be one of the below: + - * / % > < ! & ~   ^ >= <= != && == << >>	Input: according to the operator Output: the result after operated	This operator is equal to the operator of C language.
	use Resistor xxx	XXX is integer n. Delete n data from the pipe. N can larger than actual counts, which means delete all.	Input: none Output: none	
	use inductor xxx	Generate an inductor named xxx: as there's data in the pipe, it will be transferred to the inductor. As there's no data in the pipe, the data in the inductor will be transferred to the pipe.	Input: none Output: see the description	
	use capacitor xxx	Generate a capacitor named xxx: as there's data in the pipe, it will be transferred to the capacitor. As there's no	Input: none Output: see the description	

		data in the pipe, the data in the capacitor will be transferred to the pipe.		
	use Diode xxx	Simulated diode and xxx is the preset integer. If the first data equals to xxx the implement continues, otherwise this branch will be invalid.	Input: integer Output: none	
	reference capacitor xxx	quote capacitor xxx	Input: none Output: see function description	
	reference inductor xxx	quote inductor xxx	Input: none Output: see “use inductor xxx”	
	set label xxx	Set a return label named xxx	Input: none Output: none	
	goto label xxx		Input: none Output: none	
	view pipe	Show the data of the current execution pipe in running window. It won't change the data.	Input: none Output: none	
	input text [xxx]	Start a sub-dialog and ask the user to input a string. XXX is the prompt message of optional dialogs.	Input: none Output: string	
	input num [xxx]	Start a sub-dialog and ask the user to input a number. XXX is the prompt message of optional dialogs.	Input: none Output: integer or float	
	wait second xxx	Let current dialog wait for xxx second and xxx is floating number.	Input: none Output: none	
	create table xxx	Creat a data table named xxx.	Input: none Output: none	
	focus table xxx focus table	Set current focus data table. If xxx is not static,	Input: see function description	



		then use the first string in the execution pipe as the name of the table.	Output: none	
	import data	Import the data of current execution pipe into the focus data table.	Input: none Output: empty the implement pipe.	
	export data	Import the data of current focus data table into the execution pipe.	Input: none Output: see function description	
	insert data	Insert the data of current execution pipe into the focus data table and the first data of execution pipe must be an integer and pointed to the inserted index.	Input: see function description Output: empty the implement pipe.	
	modify data	Modify a line in current focus data table with the data of execution pipe and the first data of execution pipe must be an integer and pointed to the inserted index.	Input: see function description Output: empty the implement pipe.	
	get data	Put the data of certain row of current data table into the execution pipe. The row number is pointed by the first data of execution pipe which must be an integer.	Input: see function description Output: see function description	
	remove data	Delete the data of certain row of current data table. The row number is pointed by the first data of execution pipe which must be an integer.	Input: see function description Output: none	
	get size	Get the number of row of current data table and put it into the execution pipe.	Input: none Output: integer	
	close table	Close the current focus data table.	Input: none Output: none	

	use logic xxx	Use the temporary logic named xxx.	Input: none Output: none	
	focus logic xxx	Set current focus logic named xxx. If it isn't given statically, then draw the first data in execution pipe as its name, which must be a string.	Input: see function description Output: none	
	insert logic xxx	Insert temporary logic named xxx into current focus logic project.	Input: none Output: none	
	remove logic	Remove the logic branch where this order located from current focus logic project.	Input: none Output: none	
	get date	Get current date.	Input: none Output: string	
	get time	Get current time.	Input: none Output: string	
	output info	Fetch the first data of execution pipe and print show it on the running window. It must be a string.	Input: string Output: none	
	start object xxx	Start a exterior object named xxx and set it as focus.	Input: none Output: none	
	name object xxx	Name the focus object. If the name can't be given statically, then fetch the first data of execution pipe as the name and it must be a string.	Input: see function description Output: none	
	focus object xxx	Set focus object. If the name can't be given statically, then fetch the first data of execution pipe as the name and it must be a string.	Input: see function description Output: none	
	use object	Use current focus object.	Input: see the user manual of this object	

			Output: see the user manual of this object	
	close object	Close current focus object	Input: none Output: none	
	Get object document	Get the user document of current focus object.	Input: none Output: string	
inner independent command	think logic xxx	The brain goes into thinking mode and xxx is the name of the logic to be thought.	Input: none Output: none	
	Run	Continue the implementation of current logical task.	Input: none Output: none	
	Debug	The brain goes into debug mode.	Input: none Output: none	
	Stop	Stop the implementation of current logical task.	Input: none Output: none	
	Pause	Pause the implementation of current logical task.	Input: none Output: none	
	Step	Implement the current logical task step by step.	Input: none Output: none	
	close dialog	Close current dialog.	Input: none Output: none	
inner dependent command	learn token xxx	Memorize xxx as a token.	Input: none Output: none	
	learn pronoun xxx	Memorize xxx as a pronoun.	Input: none Output: none	
	learn adjective xxx	Memorize xxx as an adjective.	Input: none Output: none	
	learn numeral xxx	Memorize xxx as a numeral.	Input: none Output: none	
	learn verb xxx	Memorize xxx as a verb.	Input: none Output: none	
	learn adverb xxx	Memorize xxx as an adverb.	Input: none Output: none	
	learn article xxx	Memorize xxx as an article.	Input: none Output: none	
	learn preposition xxx	Memorize xxx as a preposition.	Input: none Output: none	
	learn conjunction xxx	Memorize xxx as a conjunction.	Input: none Output: none	

	learn interjection xxx	Memorize xxx as an interjection.	Input: none Output: none	
	learn noun xxx	Memorize xxx as a noun.	Input: none Output: none	
	learn text xxx;	Memorize text xxx. If the text isn't given statically, then the user will be asked to input a text.	Input: none Output: none	
	learn logic xxx	Memorize a temporary logic xxx.	Input: none Output: none	
	learn object xxx	Memorize an exterior object xxx.	Input: none Output: none	
	learn action xxx	Memorize temporary logic xxx as a behavior and the user will be asked to input behavior phrase and user manual.	Input: none Output: none	The words necessary for behavior phrases must be learnt in advance.
	Find xxx	Find key word xxx in memory.	Input: none Output: none	
	find logic xxx	Find key word xxx in memorized logics.	Input: none Output: none	
	find object xxx	Find key word xxx in memorized objects.	Input: none Output: none	
	set memory size	Set the memory size to be searched.	Input: none Output: none	