

## Introduction

The sameness of interface for driving cars does not limit manufacturers in designing newer and better engines. The distributed operating system Z47 provides a uniform view of all platforms without limiting their features. Through Z++ language, Z47 Processor brings out the best of any platform. After all driving from state to state feels pleasant only when the traffic signs mean the same thing.

Scientifically designed and well-tested linguistic abstractions speed up the building of reliable distributed applications. Nonetheless, distributed applications need automated Internet tools to facilitate their cooperation among heterogeneous nodes. **The Z++ Internet Server** provides that help.

The **Z++ Internet Server** plays important roles in distributed computing models of Z++, component-orientation, database interactions and so on. In this reference we explain each role and its usage.

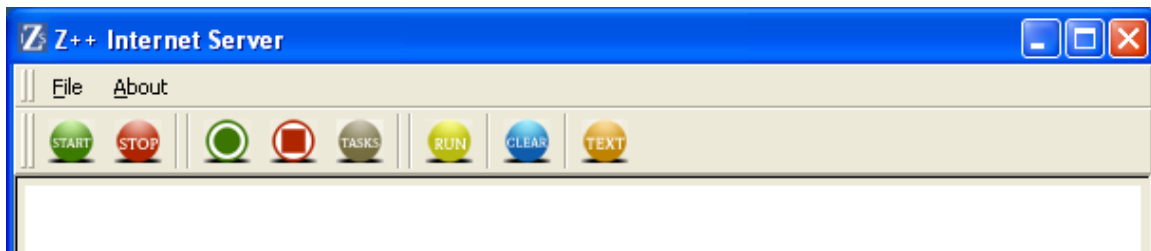
The product downloaded as **Z++ Internet Server** includes two separate servers: the **Internet Server** and the **Z47 Listening** mode (**Z47L**) Server.

The **Internet Server** deals with **Remote Procedure Call (RPC)**, **Autonomous Agents**, **Database Proxy**, **PHP requests** and **Component Requests**. The **Z47L** is for executing multiple Z++ applications on a single processor, in particular for applications using **tell/hear signaling**. The tell/hear signaling provides a solution for **Web Services** among other things.

For Linux (UNIX), the package also includes **Z++ UNIX Domain Server** for safe PHP requests. The **UNIX Domain Server** does not open any ports.

**The Internet Server uses port 14747, and the Z47L uses port 14797.**

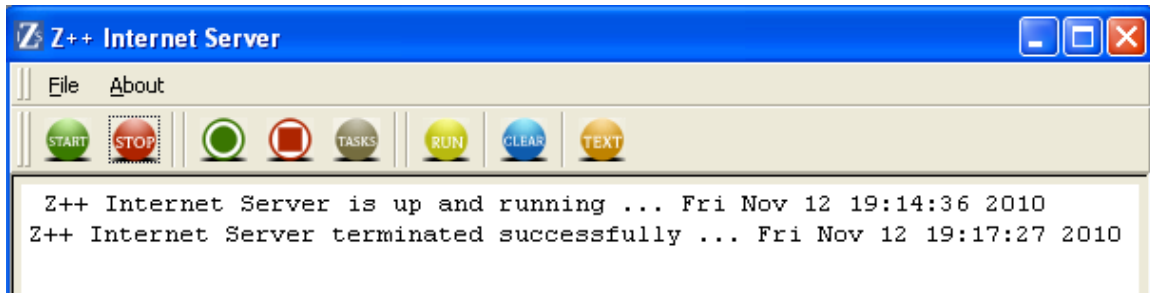
**Z++ Internet Server** provides a few buttons for starting and stopping servers and other operation that we briefly explain here.



The **Internet Server** is started, by clicking the left-most green button labeled **START**, and stopped by clicking the red button to its right, labeled **STOP**.



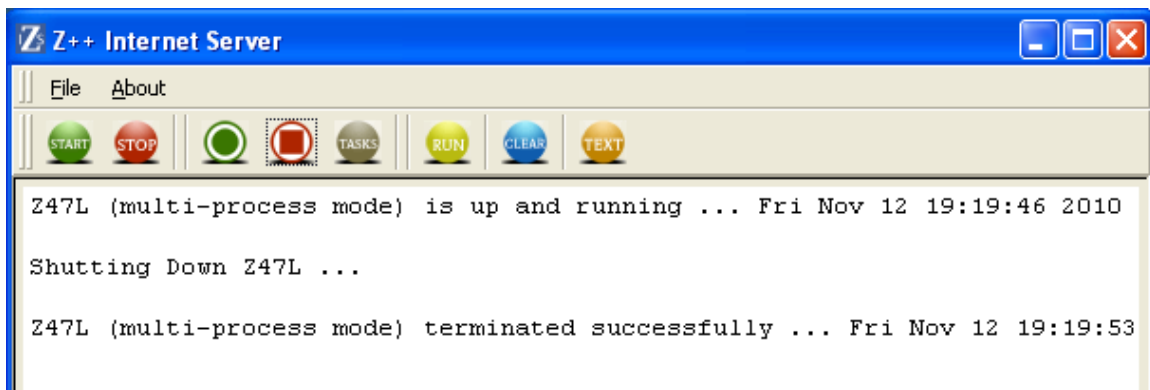
The Server prints following messages when started and stopped.



The **Z47L Server** is started by clicking the green button and stopped by clicking the red button shown below.

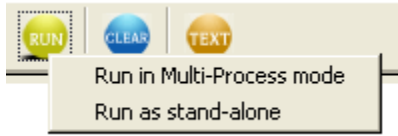


Z47L prints the following messages at startup and shutdown. In particular if there are processes still running it will kill them prior to shutdown.



The button labeled **RUN** lets you load and execute a Z++ application. It gives you a choice for the server to run the application. The **Internet Server** runs an application as a single process. That is, it runs each application on a separate Z47 Processor. On the other hand, **Z47L Server** runs all loaded applications on a **single** Z47 Processor. You can run programs on both servers simultaneously.

In figure below, Run in Multi-Process mode means load the program on **Z47L**. Run as stand-alone loads the program on the **Internet Server**.



Applications executing on **Z47L** are Z47 processes. The button labeled **TASKS** lists the executing processes and allows the killing of individual applications.

**Remark.** **Z47L** is intended for applications with Graphical User Interface. For console applications **Z47L** will write the output to its window but there is no way to enter console input.

On Windows, the application is started like any other installed program, from the start button on the task bar. On Linux, install copies **Z47Interface** to “/usr/local/bin/”. Normally you would run the program from a consol, in the background by adding the symbol & to the end of the command “**Z47Interface&**”.

On Windows, request outputs to the Internet Server go to the window of Z47Interface, while on Linux the output go to the consol from which it was started. For a commercial or future version these must go to a log file.

**Important.** On Linux, the command **Z47Interface** must be given as **root**. The servers need to open and bind sockets.

## **Table of Contents**

### The Internet Server

Database Proxy

Component-oriented Development

Component Server

Agent Server

An Agent Example

PHP Server

An example component

PHP example

Explaining the PHP script

Setting up PHP Extension

### Z47 Listening Mode

An Example of entire signals

EntryCaller.zpp

EntryOne.zpp

EntryTwo.zpp

Tell and Hear signaling

InternalTell.zpp

A Remote Telling Example

TellHear.zpp

## The Internet Server

The **Internet Server** is an integral piece of Z++ technology. It uses Z47 Processor to provide various services that we explain in the following sections.

### Database Proxy

**The Z++ language as a whole is one and the same for all platforms. The sameness includes the standard and user-developed static and dynamic libraries.**

Handheld devices are evolving and database libraries are too large to fit on a handheld device. Consequently, vendors do not provide a version of their libraries for any handheld device. Furthermore, data arriving from a query can easily overwhelm these devices. The **Database Proxy** solves these problems while keeping the Z++ language identically the same from a desktop to the tiniest device.

Z++ database statements are object-oriented extensions of SQL standard allowing program objects and database entities, like tables and records, to be intermixed. Z++ standard library includes two classes for automating the starting of a database session, carrying out transactions and finally ending the session. The classes and their use are extensively illustrated in **Z++ Language Reference**. Here, we discuss two parameters of the `class databaseType` that pertain to handheld and mobile devices.

The constructor of `databaseType` takes up to seven parameters. However, only the first three are required and the rest use default values. Below is an example of constructing an instance `Dbase` of library `class databaseType`.

```
databaseType Dbase(IP-address, // address of database server
                  port-number,
                  database-name, // database for transactions
                  "",           // server parameter if needed
                  _Database_Kind_Mysql, // vendor of database
                  10,          // number of records to fetch
                  URL to Z++ Internet Server);
```

The last two arguments are specifically needed for a handheld platform: the fetch size and the IP address of a **Z++ Internet Server** for acting as a database proxy. Obviously, database applications on desktops may need to use these parameters just as well.

Specifying a URL for a **Z++ Internet Server** tells the Z47 Processor to send all database requests to that server. Note that **Z++ Internet Server** is not the same as the Database Server with which transactions will be carried out. The **Database Proxy** of **Z++ Internet Server** will perform the transactions with the database server on behalf of its client and deliver the results to the client.

This function of the **Database Proxy** relieves a handheld device or a desktop application from direct interaction with a database server.

The fetch size of 0 will deliver the entire result of a query to the client in one transaction. In the above example we are using a fetch size of 10. In this case, the **Database Proxy** will only deliver 10 records at a time to its client. Note, however, that the fetch size for limiting the delivery of records can be used without going through the **Database Proxy**.

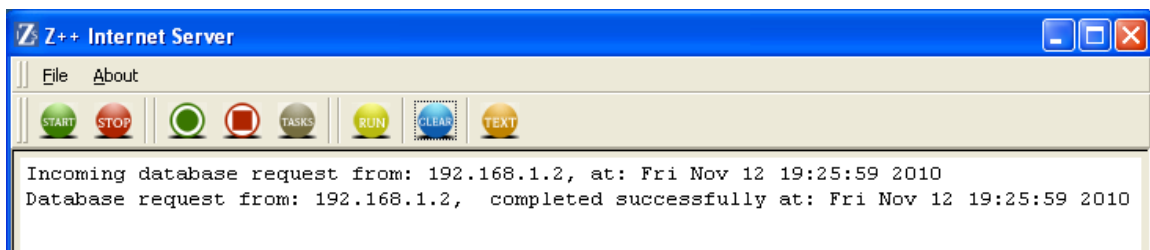
Limiting the fetch size alleviates the possibility of overwhelming the client, in particular when the client is a handheld device.

What is more important here is the sameness of Z++ database statement for all platforms regardless of the size of the device powering them. This eliminates the unnecessary practice of having to write different database statements depending on the target platform.

Indeed in most cases engineers must also engage in writing specialized scripts in addition to the application they are developing. The purpose of such scripts is to translate database statements of the language for interaction with different database management systems. **Platform-independent languages that need the assistance of such scripts belong to the stone-age era of software development.**

**Z++ database statements are intuitive object-oriented extensions of corresponding familiar SQL statements.**

The Database Proxy reports database activities as shown below.



## Component-oriented Development

We are in the era of huge enterprise applications. The development of such applications requires various specializations such as artificial intelligence, database, Internet and so on. Software developing companies compile large teams of engineers with various levels of experience with the hope that training and self-study will prepare the engineers to fill the expert needs.

Software development will become a more reliable process when each area of specialization, is taken on by competing dedicated software companies. After all going to a heart surgeon for a lung problem may not work quite well. But even as things are, large software companies can break down their teams and locate them at a distance from one another. Each team can engage with aspects of the software in accordance to its specialization and without interferences from other teams.

**Z++ language is inherently component-oriented.** A large Z++ program is composed of smaller more specialized programs, called components. In the context of Z++ a component is only a relative term. There is no distinction between a stand-alone program and a component. **The difference is in how a Z++ program is used in a development.** Thus, a program used in composing a larger program is called a component.

Examples of component-oriented development are included in **Z++ Visual**, and explained in **Z++ Language Reference**, as well as in **White Papers**.

## Component Server

For clarity, when a larger program is composed of smaller components, we will refer to the larger program as the **caller**. With that in mind, a caller may simply find all its components on the same node as it is running. **The components will simply become child processes of the caller.** In this case there is no need for the **Internet Server**.

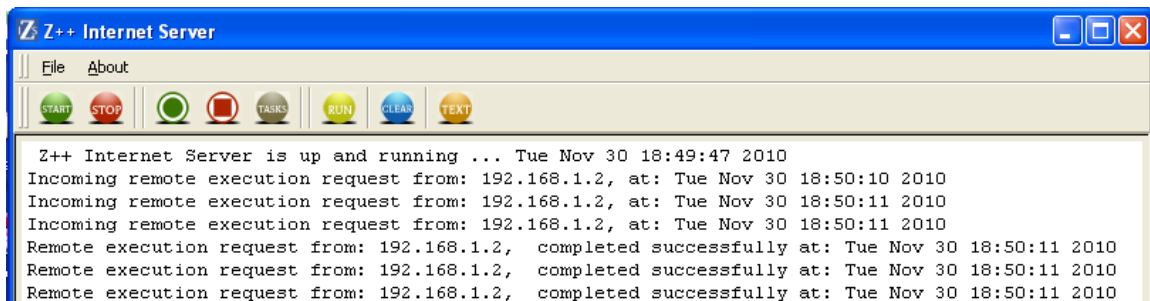
When the caller needs components that reside on remote nodes, it has to send a request to a **Z++ Internet Server**. The caller can either request to download the component from the server, or have the server execute the component remotely. In the former case the component becomes a local child process of the caller just as if the component resided on the same node as the caller is executing. In the latter case however, the component becomes a remote child process of the caller. **Keep in mind that Z47 is a distributed operating system.**

When the requested component is a **remote child process**, communication between the caller program and the component follows the pattern of Remote Procedure Call. However, the engineer is not aware of any of the complexities that we have described in that he or she does not write any relevant code. All of this is taken care of by the Z++ compiler.

The **Component Server** of **Z++ Internet Server** is the recipient of a request for a component. As requested, it will either send the entire component to the caller (the client), or will start up the component for interaction with the caller. In the latter case, the **Component Server** will also respond to the remote procedure calls coming from the caller. The **Component Server** is responsible for initialization and termination of the component, as well.

The **Z++ Internet Server** in its role as **Component Server** facilitates component-oriented development by specialized teams. Each team is able to interact with components of other teams while continuing its development at its own site. After the completion of development, all the components can reside on a single node, or dispersed as needed.

Below is output of the **Z++ Internet Server** responding to a remote client.



```
Z++ Internet Server is up and running ... Tue Nov 30 18:49:47 2010
Incoming remote execution request from: 192.168.1.2, at: Tue Nov 30 18:50:10 2010
Incoming remote execution request from: 192.168.1.2, at: Tue Nov 30 18:50:11 2010
Incoming remote execution request from: 192.168.1.2, at: Tue Nov 30 18:50:11 2010
Remote execution request from: 192.168.1.2, completed successfully at: Tue Nov 30 18:50:11 2010
Remote execution request from: 192.168.1.2, completed successfully at: Tue Nov 30 18:50:11 2010
Remote execution request from: 192.168.1.2, completed successfully at: Tue Nov 30 18:50:11 2010
```

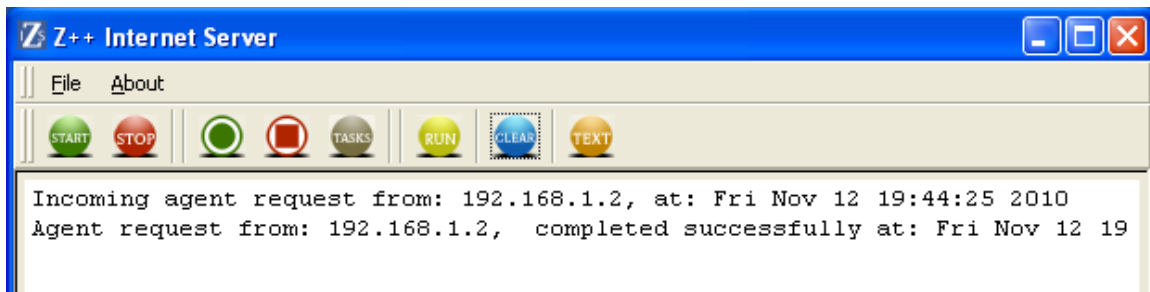
## Agent Server

Creation and maintenance of Z++ technology needs experts on each platform in order to provide an **abstract unified view of all platforms**. The self-contained design of Z47 Processor eliminates technical hurdles for its presence on any platform. However, Z47 Processor is not yet universally available. For that reason, **Agent Server** is currently distributed for desktops but not for handheld devices.

The **Agent Server** accommodates the arrival of an **Autonomous Agent**. A Z++ agent is a process of the distributed operating system Z47 Processor. An agent travels as a live process from node to node **maintaining its entire state as an executing process**.

The **Agent Server** begins the execution of an arriving agent as if the agent is still running on the node from which it came. The state of objects, the point of execution and opened scopes are all preserved during the teleport of an agent.

The **Agent Server** reports incoming agent requests as shown below.



## An Agent Example

In the example [ScopedTravel.zpp](#), presented in this section notice that when the agent takes off the point of execution is inside a **switch** statement, which is nested in a loop. The teleportation of the agent begins with the statement “**travel** *address*;”. The term **travel** is a Z++ language keyword that sends an executing process to the destination indicated in its *address* argument. The IP address is the address of a machine, which is running the **Z++ Internet Server**.

Notice that the **state** of the enumeration object *stateOfTravel*, and the **boolean** object *Done* are important for the correct execution of the code, before and after **travel**.

Incrementing an enumeration object changes the value of the object to its successor. Thus, “*stateOfTravel++*,” changes the value of the object *stateOfTravel* from the enumeration literal *\_travel\_Before* to the next literal *\_travel\_After*.

The agent, on the machine that it is executing prior to travel, writes to the file *TravelBefore.txt* that it has not yet traveled. When it arrives at its destination, writes to the file *TravelAfter.txt* that it has reached its destination.

```

//ScopedTravel.zpp

#include<stream.h>
using namespace fileSpace;

enum TravelStates {_travel_Before, _travel_After, _travel_End};

string outDir = "C:/"; // directory for creating file

entry void main(void)

    string info;
    string outname;
    string address = "192.168.1.2"; // use correct IP address here

    TravelStates stateOfTravel = _travel_Before;
    boolean Done = False;

    do
        switch(stateOfTravel)

            case _travel_Before:
                info = "We have not traveled, yet.\n";
                outname = outDir + "TravelBefore.txt";

            case _travel_After:
                info = "Traveled to our destination.\n";
                outname = outDir + "TravelAfter.txt";
                travel address; // Take off to destination

            else Done = True; // At destination

        endswitch;

        if (!Done)
            stateOfTravel++; // Increment to the successor
            FileStreamType outfile;
            outfile.setOutput();
            outfile.open(outname);
            outfile << info;
            outfile.close();

        endif;

    enddo (Done);

end;

```

## PHP Server

The **PHP Server** is part of the **Component Server**. PHP requests are served in a manner similar to remote processes of caller. PHP uses a simple pattern to interact with Z++, which we will describe. We will also explain the familiar set up for PHP scripts to interact with PHP Server.

The **PHP Server** is available for Linux and comes in two flavors. PHP scripts can either go through the **Internet Server** or **Unix Domain Server**, which does not use any port. The **Unix Domain Server** is quite significant for enterprise solutions.

Generally, solutions are developed for internal use within an enterprise, and much of the code is duplicated in PHP scripts. Z++ components unify the two development activities and reduce them to a single development. The same component that is used for internal purposes can also be used by, PHP scripts. However, Z++ is a far more expressive language for developing complex end-to-end distributed enterprise solutions. Since **Unix Domain Server does not open any ports**, there is no threat of abuse of open ports.

The Z++ components for use via the **Unix Domain Server** must reside on the same node as the server. But that is essentially what one would like to do, any way. On the other hand, PHP scripts that use the **Internet Server** can reach Z++ components on any node. Note that the **Z++ component is one and the same regardless of how it is reached**.

## An example component

In order to explain the complete process, we start with a Z++ sample program. The following sample is a plain component oblivious of how it will be used. For simplicity, the component consists entirely of **entry** points. Recall that the set of **entry** points of a component comprises its boundary with the outside world. Callers interact with a component by invoking its **entry** points.

The general pattern of each **entry** point in the following example is to open a file in the tmp directory, write something to the file and close it. The difference is in the signature of the **entry** points for purposes of illustration.

```
//PhpLinuxSample.zpp

#include<iostream.h>
using namespace ioSpace;

#include<stream.h>
using namespace fileSpace;

FileStreamType outfile;

entry void main(void)
    output << "Hello World!\n";
    outfile.setOutput();
    string filename = "/tmp/phpRemoteMain.txt";
    outfile.open(filename);
    outfile << "Called from remote PHP site.\n";
    outfile.close();
    output << "Good-bye World!\n";

end;

entry void trial(int arg)
    outfile.setOutput();
    string filename = "/tmp/phpRemoteTrial.txt";
    outfile.open(filename);
    outfile << "Called from remote PHP site.\n";
    outfile << "Argument is : " << arg << '\n';
    outfile.close();

end;

entry void multiarg(long height, int age, double weight, char gender)
    outfile.setOutput();
    string filename = "/tmp/phpRemoteMultiarg.txt";
    outfile.open(filename);
    outfile << "Called from remote PHP site.\n";
    outfile << "Height is : " << height << '\n';
    outfile << "Age is : " << age << '\n';
    outfile << "Weight is : " << weight << '\n';
    outfile << "Gender is : " << gender << '\n';
    if (gender == 'M') outfile << "Gender is : Male.\n";
    else outfile << "Gender is : Female.\n";
```

```

        endif;
        outfile.close();
end;

entry int returnArg(int arg)
    outfile.setOutput();
    string filename = "/tmp/phpRemoteReturnArg.txt";
    outfile.open(filename);
    outfile << "Called from remote PHP site.\n";
    outfile << "Argument is : " << arg << '\n';
    outfile.close();
    return 3*arg;
end;

entry double stringarg(string name, int age,
                        double weight, string last)

    outfile.setOutput();
    string filename = "/tmp/phpRemoteStringarg.txt";
    outfile.open(filename);
    outfile << "Called from remote PHP site.\n";
    outfile << "Name is : " << name << '\n';
    outfile << "Lastame is : " << last << '\n';
    outfile << "Age is : " << age << '\n';
    outfile << "Weight is : " << weight << '\n';
    outfile.close();
    return weight + 47;
end;

entry string stringReturn(string name)
    outfile.setOutput();
    string filename = "/tmp/phpRemoteStringReturn.txt";
    outfile.open(filename);
    outfile << "Called from remote PHP site.\n";
    outfile << "Name is : " << name << '\n';
    outfile.close();
    return "Jackson";
end;

```

To use this program, first you compile it to a Z++ executable, which we shall call `Sample.zxe` here. This filename is used in the PHP script presented in the next section.

## PHP example

Below is the PHP script using the Z++ component presented in previous section. Detailed explanations about invoking the **entry** points of Z++ component follow the script.

Note that except for starting up the Z++ component the rest of the code remains the same. The difference in start up is how `zpp_url` is initialized. If the IP address is provided the script will contact the **Z++ Internet Server**. On the other hand, if instead the symbol `@` is used, the script will engage with **Z++ Unix Domain Server**.

The objects sent to an **entry** point are hard coded for simplicity. These objects are usually obtained via an HTML form or similar mechanism prior to invoking an **entry** point.

```
<?php
    echo '<p>Hello World</p>';

// You will need to uncomment one of the following lines.
// The first line is for working with Z++ Internet Server
// (uses port 14747), and the second line is for
// Z++ Unix Domain Server which does not use any port.

// In first line you will need to replace the xx.xx.xx.xx
// with IP address of the machine that has the file
// Sample.zxe (the machine running the Z++ Internet Server).
// For the second line, the file must be on local machine.
// The @ symbol indicates that to the Z++ PHP extension.

// Next line for using Z++ Internet Server
    //$zpp_url = 'xx.xx.xx.xx/path-to/Sample.zxe';

// Next line for using Z++ Unix Domain Server
    //$zpp_url = '@/path-to/Sample.zxe';

    echo '<p>Loading Z++ Module</p>';

    $zpp_socket = (int) php_zpp_zxe_load($zpp_url);

    if ($zpp_socket == 0) {
        echo '<p>Unable to load Z++ Module</p>';
    }

    else { // Invoking entry void main(void);
        $arguments = array();
        $arguments[0] = "main";
        $arguments[1] = "v/v";

        printf("Calling entry : %s", $arguments[0]);
        echo'<br>';
        if (!php_zpp_entry($zpp_socket, $arguments)) {
            printf("Failed calling entry : %s", $arguments[0]);
            echo'<br>';
        }
        else { // Invoking entry void trial(void);
```

```

$arguments = array();
$arguments[0] = "trial";
$arguments[1] = "i/v";
$arguments[2] = 25;

printf("Calling entry : %s", $arguments[0]);
echo'<br>';
if (!php_zpp_entry($zpp_socket, $arguments)) {
    printf("Failed calling entry : %s", $arguments[0]);
    echo'<br>';
}
else { // entry void multiarg(long, int, double, char);
    $arguments = array();
    $arguments[0] = "multiarg";
    $arguments[1] = "lidc/v";
    $arguments[2] = 297;
    $arguments[3] = 25;
    $arguments[4] = 89.74;
    $arguments[5] = "M";

    printf("Calling entry : %s", $arguments[0]);
    echo'<br>';
    if (!php_zpp_entry($zpp_socket, $arguments)) {
        printf("Failed calling entry : %s", $arguments[0]);
        echo'<br>';
    }
    else { // Invoking entry int returnArg(int);
        $arguments = array();
        $arguments[0] = "returnArg";
        $arguments[1] = "i/i";
        $arguments[2] = 25;
        $arguments[3] = 555;

        printf("Calling entry : %s", $arguments[0]);
        echo'<br>';
        if (!php_zpp_entry($zpp_socket, $arguments)) {
            printf("Failed calling entry : %s", $arguments[0]);
            echo'<br>';
        }
    }
}
// Invoking entry double stringarg(string, int, double, string);
else {
    printf("value received was : %d", $arguments[3]);
    echo'<br>';
    $arguments = array();
    $arguments[0] = "stringarg";
    $arguments[1] = "sids/d";
    $arguments[2] = "Jack";
    $arguments[3] = 25;
    $arguments[4] = 89.74;
    $arguments[5] = "Jackson";
    $arguments[6] = 0;

    printf("Calling entry : %s", $arguments[0]);
    echo'<br>';
    if (!php_zpp_entry($zpp_socket, $arguments)) {
        printf("Failed calling entry : %s",
            $arguments[0]);
    }
}

```



## Explaining the PHP script

The line

```
$zpp_socket = (int) php_zpp_zxe_load($zpp_url);
```

loads the Z++ component `Sample.zxe` and starts it as a Z47 process. The **Component Server** of the **Internet Server** or the **UNIX Domain Server** will then await calls to the **entry** points of the component.

There is a simple pattern for invoking **entry** points, as follows.

- Declare a PHP array (called `arguments` in the script).
- Put name of **entry** in first cell of the array.
- Put the signature of the **entry** in the second cell.
- Put the call arguments in following cells.
- If the call returns a value, initialize one final cell, with 0 or anything else.
- Call the **entry** as shown below

```
php_zpp_entry($zpp_socket, $arguments)
```

The library function `php_zpp_entry` returns false on failure.

The signature put in the second cell is a string constructed as follows. The slash is a separator between arguments and the return type. The type of each argument and the return object takes one character using the following table.

```
c ----- char
t ----- short
i ----- int
l ----- long
f ----- float
d ----- double
s ----- string
v ----- void
```

For instance, the string `"lfdc/v"` means that, the **entry** point will receive a **long**, an **int**, a **double** and a **char** in that order, and does not return a value.

At the end, the library function `php_zpp_zxe_free` terminates the process executing the code of the Z++ component.

## Setting up PHP Extension

The set up is simple, familiar and quick. The purpose of setup is to enable PHP scripts to use Z++ as was shown in the example in the previous section.

When you unzip your PHP package (downloaded from ZHMicro) you will find just a few files. Follow these steps.

- Create a subdirectory called “zpp” in the “ext” directory of PHP.
- Copy three files: “zpp.c”, “php\_zpp.h” and “config.m4” to “zpp” directory.
- Go to “zpp” directory and issue the following commands to create “zpp.so”.

```
phpize
./configure --enable-zpp
make clean all
```

- Copy **zpp.so** to the directory that **php.ini** expects.
- Add the following line to **php.ini** in the section “Dynamic Extensions”.

```
extension=zpp.so
```

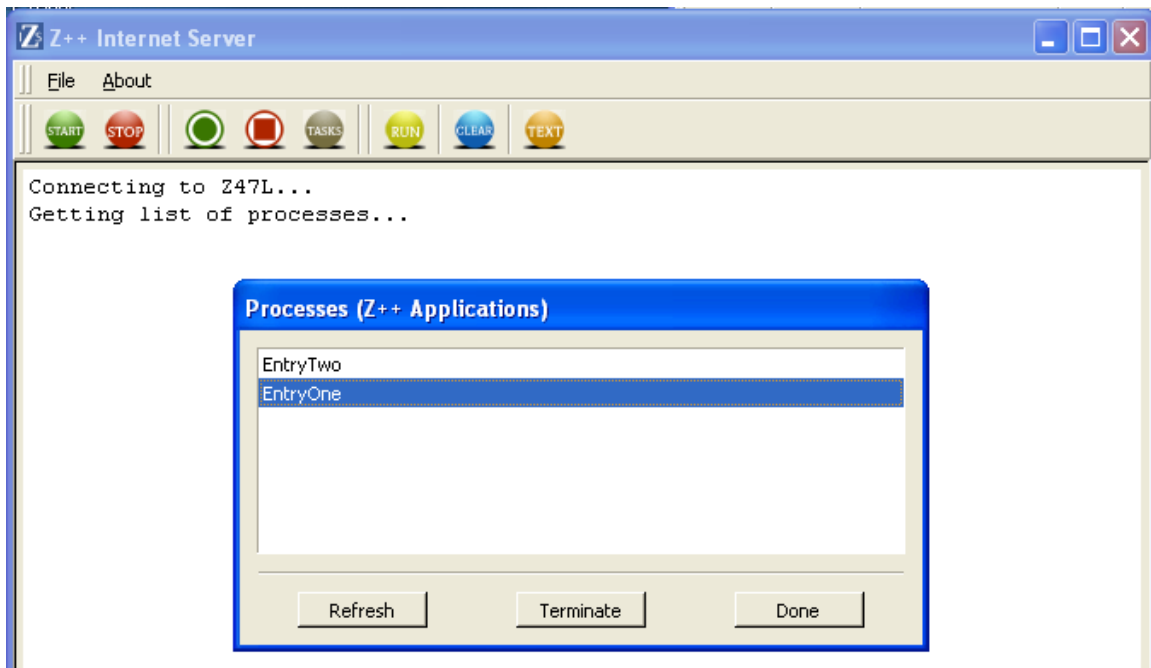
You are now ready to use Z++ in PHP scripts. However, the Z++ extension will become available after you restart your **Apache Server**.

## Z47 Listening Mode

The Z47 Listening Mode Server, or **Z47L** is an interface to the Z47 distributed operating system. On each node, you can run all Z++ applications on a single Z47 processor. These applications, or Z47 processes, can communicate to one another, and to remote processes.

In Introduction we saw how to start and stop **Z47L**. We also discussed how to load an application on **Z47L**, by selecting “Run in Multi-Process mode” from Run button or menu item.

In this chapter we will present some examples of tell/hear and other forms of signals. Before proceeding with examples, let us mention the use of button **Tasks**. When you click the Tasks button, or choose it from menu options, **Z47L** lists the applications currently executing. You can select a process and terminate (kill) it.



When you terminate a process, **Z47L** sends `_INTERRUPTION_TerminateProcess` signal to the process. The process terminates and the following message appears on the screen. We have chosen to terminate the application called EntryOne.

Unhandled exception (EntryOne) ==> `_INTERRUPTION_TerminateProcess`

To close the popup of list of processes click the button **Done**.

## An Example of entire signals

[Z++ Language Reference Manual](#) discusses various types of Z++ signals. In this section we are going to illustrate Process and Node Entire Signals. The main focus is on node entire signals. These signals are intended for communication among processes running on the same node. The distributed (remote) inter-process signals, tell/hear, will be illustrated in the next section.

In the following example, [EntryCaller.zpp](#), there are three types of user-defined signals. In Z++ any kind of user-defined signals are derived from appropriate system signal category. The simplest kind of signal is derived from [signalEventType](#) system signals. These signals should have the prefix “[\\_SIGNAL\\_](#)”. We have one user-defined signal in this category, [\\_SIGNAL\\_Thread\\_ended\\_1](#). These signals can be used for inter-thread communication within a process, without registration. Only one thread will be able to catch this kind of signal.

Sometimes we wish to send a signal to several threads of a process. In Z++ signals that can be caught by more than one thread or process are called **Entire**. The reason for naming is that all threads or processes that register to receive these signals will in fact receive it. Entire signals for inter-thread communication within a process are derived from [threadEntireEventType](#), and should have the prefix “[\\_THREAD\\_ENTIRE\\_](#)”. These signals are also called process-bounded because they cannot go from one process to another.

The focus of our example is node-bounded entire signals. These are intended for communication among processes running on a single node, or same [Z47L](#). User-defined signals of this kind are derived from [processEntireEventType](#), and identified with the prefix “[\\_PROCESS\\_ENTIRE\\_](#)”.

Catching any kind of entire signal requires registration through Z++ [accepts\(...\)](#) specification. In the following example notice how the global thread and the task thread register the signals they wish to be delivered to them.

Node-bounded signals (process-entire signals) must be registered by an [entry](#) point of the application. So, we see this in the examples [EntryOne.zpp](#) and [EntryTwo.zpp](#) that follow [EntryCaller.zpp](#).

**Remark.** These examples are packaged with [Z++ Visual](#). To run them, load the executables [EntryOne.zxe](#) and [EntryTwo.zxe](#), before loading [EntryCaller.zxe](#). This is because sending unregistered entire signals will be lost.

The program output is shown in red. Start with main [entry](#) point and follow the output as you go along. The output essentially illustrates what the program is doing.

```

// EntryCaller.zpp

#include<iostream.h>
using namespace ioSpace;

#include<exception.h>
using namespace exceptionSpace;

enum MyOwnSignals : signalEventType {
    _SIGNAL_Thread_ended_1
};

enum MyEntireSignals : threadEntireEventType {
    _THREAD_ENTIRE_Signal_1,
    _THREAD_ENTIRE_Signal_2,
    _THREAD_ENTIRE_Signal_3
};

enum MyEntrySignals : processEntireEventType {
    _PROCESS_ENTIRE_Signal_1,
    _PROCESS_ENTIRE_Signal_2
};

////////////////////////////////////
// FirstHandlerOne() fires when main sends _THREAD_ENTIRE_Signal_1.
// -----

task One

    accepts(_THREAD_ENTIRE_Signal_1); // register signals

    void FirstHandlerOne(void)<_THREAD_ENTIRE_Signal_1>;

public:
end;

// -----

void One::FirstHandlerOne(void)
    output << "EntryCaller, Inside FirstHandlerOne(void). Sending
signal _THREAD_ENTIRE_Signal_2.\n";
    signal <- _THREAD_ENTIRE_Signal_2;
    output << "EntryCaller, Inside FirstHandlerOne(void). Sending
signal _THREAD_ENTIRE_Signal_3.\n";
    signal <- _THREAD_ENTIRE_Signal_3;
end;

////////////////////////////////////
// Whan task handler FirstHandlerOne() sends both signals, this
// will generate the two process-entire signals on which
// EntryOne.zxe and EntryjTwo.zxe are waiting.
// -----

void FirstGlobalThread(void)<thread>
accepts(_THREAD_ENTIRE_Signal_2, _THREAD_ENTIRE_Signal_3) // register

```

```

    output << "EntryCaller, FirstGlobalThread(): waiting for
_THREAD_ENTIRE_Signal_2.\n";
    do enddo(signal ? _THREAD_ENTIRE_Signal_2);

    output << "EntryCaller, FirstGlobalThread(): waiting for
_THREAD_ENTIRE_Signal_3.\n";
    do enddo(signal ? _THREAD_ENTIRE_Signal_3);

    output << "EntryCaller, FirstGlobalThread(): sending
_PROCESS_ENTIRE_Signal_1.\n";
    signal <- _PROCESS_ENTIRE_Signal_1;

    output << "EntryCaller, FirstGlobalThread(): sending
_PROCESS_ENTIRE_Signal_2.\n";
    signal <- _PROCESS_ENTIRE_Signal_2;

    output << "EntryCaller, FirstGlobalThread(): sending
_SIGNAL_Thread_ended_1.\n";
    signal <- _SIGNAL_Thread_ended_1;
end;

////////////////////////////////////

entry void main(void)

    output << "EntryCaller: Hello World!\n";

    output << "EntryCaller: Starting task and thread\n";

    One N; // start the task
    FirstGlobalThread(); // start the global thread

    output << "EntryCaller: Sending _THREAD_ENTIRE_Signal_1 to
task.\n";

    signal <- _THREAD_ENTIRE_Signal_1;

    output << "EntryCaller: Waiting for _SIGNAL_Thread_ended_1.\n";
    do enddo(signal ? _SIGNAL_Thread_ended_1);

    output << "EntryCaller: Good-bye World!\n";

end;

```

```

// EntryOne.zpp

// -----
// When loaded will wait for EntryCaller.zxe send the signals
// _PROCESS_ENTIRE_Signal_1 and _PROCESS_ENTIRE_Signal_2
// -----

#include<iostream.h>
using namespace ioSpace;

#include<exception.h>
using namespace exceptionSpace;

////////////////////////////////////

enum MyOwnSignals : signalEventType {
    _SIGNAL_Thread_ended_1
};

enum MyEntireSignals : threadEntireEventType {
    _THREAD_ENTIRE_Signal_1,
    _THREAD_ENTIRE_Signal_2,
    _THREAD_ENTIRE_Signal_3
};

enum MyEntrySignals : processEntireEventType {
    _PROCESS_ENTIRE_Signal_1,
    _PROCESS_ENTIRE_Signal_2
};

////////////////////////////////////

task One

    accepts(_THREAD_ENTIRE_Signal_1);

    void FirstHandlerOne(void)<_THREAD_ENTIRE_Signal_1>;

public:
end;

void One::FirstHandlerOne(void)
    output << "EntryOne, Inside FirstHandlerOne(void). Sending signal
    _THREAD_ENTIRE_Signal_2.\n";
    signal <- _THREAD_ENTIRE_Signal_2;
    output << "EntryOne, Inside FirstHandlerOne(void). Sending signal
    _THREAD_ENTIRE_Signal_3.\n";
    signal <- _THREAD_ENTIRE_Signal_3;
end;

////////////////////////////////////

void FirstGlobalThread(void)<thread>
accepts(_THREAD_ENTIRE_Signal_2, _THREAD_ENTIRE_Signal_3)

```

```

    output << "EntryOne, FirstGlobalThread(): waiting for
_THREAD_ENTIRE_Signal_2.\n";
    do enddo(signal ? _THREAD_ENTIRE_Signal_2);

    output << "EntryOne, FirstGlobalThread(): waiting for
_THREAD_ENTIRE_Signal_3.\n";
    do enddo(signal ? _THREAD_ENTIRE_Signal_3);

    output << "EntryOne, FirstGlobalThread(): sending
_SIGNAL_Thread_ended_1.\n";
    signal <- _SIGNAL_Thread_ended_1;
end;

////////////////////////////////////
// process-entire signals are registered by entry points.

entry void main(void)
accepts(_PROCESS_ENTIRE_Signal_1, _PROCESS_ENTIRE_Signal_2)

    output << "EntryOne: Hello World!\n";

    output << "EntryOne: Starting task and thread\n";

    One N; // start task
    FirstGlobalThread(); // start global thread

    output << "EntryOne: Sending _THREAD_ENTIRE_Signal_1 to task.\n";

    signal <- _THREAD_ENTIRE_Signal_1;

    output << "EntryOne: Waiting for _SIGNAL_Thread_ended_1.\n";
    do enddo(signal ? _SIGNAL_Thread_ended_1);

    output << "EntryOne: Waiting for _PROCESS_ENTIRE_Signal_1.\n";
    do enddo(signal ? _PROCESS_ENTIRE_Signal_1);

    output << "EntryOne: Waiting for _PROCESS_ENTIRE_Signal_2.\n";
    do enddo(signal ? _PROCESS_ENTIRE_Signal_2);

    output << "EntryOne: Good-bye World!\n";

end;

```

```

// EntryTwo.zpp

// -----
// When loaded will wait for EntryCaller.zxe to send the signals
//     _PROCESS_ENTIRE_Signal_1 and _PROCESS_ENTIRE_Signal_2
// -----

#include<iostream.h>
using namespace ioSpace;

#include<exception.h>
using namespace exceptionSpace;

////////////////////////////////////

enum MyOwnSignals : signalEventType {
    _SIGNAL_Thread_ended_1
};

enum MyEntireSignals : threadEntireEventType {
    _THREAD_ENTIRE_Signal_1,
    _THREAD_ENTIRE_Signal_2,
    _THREAD_ENTIRE_Signal_3
};

enum MyEntrySignals : processEntireEventType {
    _PROCESS_ENTIRE_Signal_1,
    _PROCESS_ENTIRE_Signal_2
};

////////////////////////////////////

task One

    accepts(_THREAD_ENTIRE_Signal_1);

    void FirstHandlerOne(void)<_THREAD_ENTIRE_Signal_1>;

public:
end;

void One::FirstHandlerOne(void)
    output << "EntryTwo, Inside FirstHandlerOne(void). Sending signal
    _THREAD_ENTIRE_Signal_2.\n";
    signal <- _THREAD_ENTIRE_Signal_2;
    output << "EntryTwo, Inside FirstHandlerOne(void). Sending signal
    _THREAD_ENTIRE_Signal_3.\n";
    signal <- _THREAD_ENTIRE_Signal_3;
end;

////////////////////////////////////

void FirstGlobalThread(void)<thread>
accepts(_THREAD_ENTIRE_Signal_2, _THREAD_ENTIRE_Signal_3)

```

```

    output << "EntryTwo, FirstGlobalThread(): waiting for
_THREAD_ENTIRE_Signal_2.\n";
    do enddo(signal ? _THREAD_ENTIRE_Signal_2);

    output << "EntryTwo, FirstGlobalThread(): waiting for
_THREAD_ENTIRE_Signal_3.\n";
    do enddo(signal ? _THREAD_ENTIRE_Signal_3);

    output << "EntryTwo, FirstGlobalThread(): sending
_SIGNAL_Thread_ended_1.\n";
    signal <- _SIGNAL_Thread_ended_1;
end;

////////////////////////////////////
// process-entire signals are registered by entry points.

entry void main(void)
accepts(_PROCESS_ENTIRE_Signal_1, _PROCESS_ENTIRE_Signal_2)

    output << "EntryTwo: Hello World!\n";

    output << "EntryTwo: Starting task and thread\n";

    One N; // start task
    FirstGlobalThread(); // start global thread

    output << "EntryTwo: Sending _THREAD_ENTIRE_Signal_1 to task.\n";

    signal <- _THREAD_ENTIRE_Signal_1;

    output << "EntryTwo: Waiting for _SIGNAL_Thread_ended_1.\n";
    do enddo(signal ? _SIGNAL_Thread_ended_1);

    output << "EntryTwo: Waiting for _PROCESS_ENTIRE_Signal_1.\n";
    do enddo(signal ? _PROCESS_ENTIRE_Signal_1);

    output << "EntryTwo: Waiting for _PROCESS_ENTIRE_Signal_2.\n";
    do enddo(signal ? _PROCESS_ENTIRE_Signal_2);

    output << "EntryTwo: Good-bye World!\n";

end;

```

## Tell and Hear signaling

Tell/hear signals are intended for communication among processes running on different [Z47L](#). These signals deliver data as well. Thus, distributed applications or processes running on different nodes can exchange data via tell/hear signals. The types of data can be very complex Z++ types. Furthermore, the process that sends a hear signal continues with its operations. The target process will eventually catch and process the signal.

For short, we refer to tell/hear signal as simply hear signals. The sending of a hear signal is called telling, and receiving it is called hearing the signal.

Hear signals must be used on a [Z47L](#). However, in our first example we try to keep things simple. [InternalTell.zpp](#) can be executed on a regular Z47 because the signals are sent from the process to itself. Once again this is not a good use of hear signals. Instead use process-entire signals.

In the example, the thread `hearFun()` registers the hear signals it will accept and the types of data it will expect to receive along with each hear signal. The registration of hear signals is done via the `hears(...)` specification.

The `main()` **entry** point sends two different hear signals to `hearFun()`. Once by calling `tellerFun()`, and another directly.

```

// InternalTell.zpp

#include<iostream.h>
using namespace ioSpace;

#include<exception.h>
using namespace exceptionSpace;

////////////////////////////////////

enum MyOwnSignals : signalEventType {
    _SIGNAL_Thread_started,
    _SIGNAL_Thread_ended
};

enum MyTellSignals : tellSignalType {
    _TELL_telling_1,
    _TELL_telling_2,
    _TELL_telling_3
};

////////////////////////////////////
// Let us also use a namespace for type of data we are going to use
// for tell signals. This is not needed.

namespace tellSpace

struct argumentType
    int n;
    long g;

    argumentType(void);
    argumentType(int, long);
end;

argumentType::argumentType(void)
    n = 2;
    g = 89;
end;

argumentType::argumentType(int nn, long gg)
    n = nn;
    g = gg;
end;

endspace;

////////////////////////////////////
// This is the global data we will send out.

int mm = 25;
double dd = 56.12;
tellSpace::argumentType atp;

// Main calls this function to send data out.

```

```

void tellerFun(void)

    output << "(tellerFun) Sending values : " << mm << ' ' << dd <<
'\n';
    tell <- _TELL_telling_1 : mm, dd;

end;

////////////////////////////////////
// hearFun() is a global thread that will receive tell signals.
//
// A thread registers hear signals it wishes to receive via
// hears(...) specification.

void hearFun(void)<thread>
hears(_TELL_telling_1<int, double>,
      _TELL_telling_2<tellSpace::argumentType>)

// Create objects for receiving data from hear signals.

    int m;
    double d;
    tellSpace::argumentType hatp;

// Signal main that we are ready
    signal <- _SIGNAL_Thread_started;

// Wait until hear signal and data arrive
    do enddo(hear ? _TELL_telling_1 : m, d);

    output << "(hearFun) Values received from tell : " << m << ' ' <<
d << '\n';

// Wait until tell signal and data arrive
    do enddo(hear ? _TELL_telling_2 : hatp);

    output << "(hearFun) Received _TELL_telling_2.\n";
    output << "(hearFun) struct members are : " << hatp.n << ' ' <<
hatp.g << '\n';

    signal <- _SIGNAL_Thread_ended; // Inform main we are done

end;

////////////////////////////////////

entry void main(void)

    output << "Hello World!\n";

    hearFun(); // start the global thread

// Wait until thread is ready
    do enddo(signal ? _SIGNAL_Thread_started);

    tellerFun(); // send some tell data by calling a function

```

```
// Now make some data to send directly to hearFun()

tellSpace::argumentType latp(21, 512);
MyTellSignals tso = _TELL_telling_2;

tell <- tso : latp; // Send the tell signal and the data

// Wait until thread terminates
do enddo(signal ? _SIGNAL_Thread_ended);

output << "Good-bye World!\n";

end;
```

## A Remote Telling Example

In the next example we use two communicating processes. The one Z++ program can be compiled into two executable. When the define `HearModeOn` is on (uncommented) you get the application for hearing. Rename the executable to `Hear.zxe`. Then turn off the define `HearModeOn` and rebuild the program, and rename the executable to `Tell.zxe`.

For execution, first load `Hear.zxe`, and then load `Tell.zxe`. The reason for this is that if you load `Tell.zxe` first, it will send its hear signals. However, Z47 Processor does not find any process having registered the signals and informs the process. As a result, `Tell.zxe` will raise exception.

Before building the examples, you will need to set the string url to the correct IP address. This is the address of a `Z47L` on which the `Hear.zxe` will be waiting for the arrival of the hear signals. You can run both programs on the same `Z47L`. At any rate, `Tell.zxe` does not need to run on a `Z47L`, only `Hear.zxe` must run on a `Z47L`.

In the following statement,

```
tell <- _TELL_telling_4 $ url , "Hear.zxe": eVal, sVal;
```

we are sending the signal `_TELL_telling_4` to IP address at url, and to the process called `Hear.zxe`. Then, there is a colon followed by two data items. The \$ and the comma, and the colon are separators. The reason for using \$ as a separator is that, both operands the url and the name of process are optional. Thus, when one of them is present it is not clear which one is intended. Both operands are of type `string`.

```

// TellHear.zpp

#define HearModeOn

#include<iostream.h>
using namespace ioSpace;

#include<exception.h>
using namespace exceptionSpace;

////////////////////////////////////

enum MyOwnSignals : signalEventType {
    _SIGNAL_Thread_started,
    _SIGNAL_Thread_ended
};

enum MyTellSignals : tellSignalType {
    _TELL_telling_1,
    _TELL_telling_2,
    _TELL_telling_3,
    _TELL_telling_4,
    _TELL_telling_5,
    _TELL_telling_6,
    _TELL_telling_7
};

// -----

enum simpleEnum {
    _SIMPLE_enum_1,
    _SIMPLE_enum_2,
    _SIMPLE_enum_3
};

////////////////////////////////////

namespace tellSpace

struct argumentType
    int n;
    long g;

    argumentType(void);
    argumentType(int, long);
end;

argumentType::argumentType(void)
    n = 2;
    g = 89;
end;

argumentType::argumentType(int nn, long gg)
    n = nn;
    g = gg;

```

```

end;

endspace;

////////////////////////////////////

struct imbeddedType
    simpleEnum e;
    string s;

    imbeddedType(void);
end;

imbeddedType::imbeddedType(void)
    e = _SIMPLE_enum_1;
    s = "I am member of imbeddedType";
end;

////////////////////////////////////

string url = "192.168.1.2"; // Server IP

int mm = 25;
double dd = 56.12;
tellSpace::argumentType atp;

////////////////////////////////////

#if HearModeOn

void hearFun(void)<thread>
hears(_TELL_telling_1<int, double>,
_TELL_telling_2<tellSpace::argumentType>,
_TELL_telling_3,
_TELL_telling_4<simpleEnum, string>,
_TELL_telling_5<imbeddedType>)

    int m;
    double d;
    tellSpace::argumentType hatp;

// Hearing _TELL_telling_1 with simple numeric data

    do enddo(hear ? _TELL_telling_1 : m, d);
    output << "(hearFun) Values received from tell : " << m << ' ' <<
d << '\n';
    output.flush();

// Hearing _TELL_telling_2 with simple structure

    do enddo(hear ? _TELL_telling_2 : hatp);
    output << "(hearFun) Received _TELL_telling_2.\n";
    output << "(hearFun) struct members are : " << hatp.n << ' ' <<
hatp.g << '\n';
    output.flush();

// Hearing _TELL_telling_3 without data

```

```

do enddo(hear ? _TELL_telling_3);
output << "(hearFun) Received _TELL_telling_3, without data.\n";

// Hearing _TELL_telling_4 with enum and string

simpleEnum eVal;
string sVal;

do enddo(hear ? _TELL_telling_4 : eVal, sVal);
output << "(hearFun) Received _TELL_telling_4.\n";
output << "(hearFun) enum is : " << [eVal] << '\n';
output << "(hearFun) string is : " << sVal << '\n';
output.flush();

// Hearing _TELL_telling_5 with imbedded enum and string

imbeddedType ibt;

do enddo(hear ? _TELL_telling_5 : ibt);
output << "(hearFun) Received _TELL_telling_5.\n";
output << "(hearFun) struct imbeddedType enum member is : " <<
[ibt.e] << '\n';
output << "(hearFun) struct imbeddedType string member is : " <<
ibt.s << '\n';
output.flush();

// Inform main we are done ====

signal <- _SIGNAL_Thread_ended;

end;

////////////////////////////////////

entry void main(void)

output << "Hearer: Hello World!\n";

hearFun();
do enddo(signal ? _SIGNAL_Thread_ended);

output << "Hearer: Good-bye World!\n";

// -----
#else // HearModeOn
// -----

entry void main(void)

output << "Teller: Hello World!\n";

tell <- _TELL_telling_1 $ url: mm, dd;

// -----

tellSpace::argumentType latp(21, 512);

```

```
MyTellSignals tso = _TELL_telling_2;
tell <- tso $ url: latp;

// -----

tell <- _TELL_telling_3 $ url;

// -----

simpleEnum eVal = _SIMPLE_enum_2;
string sVal = "String from main.";

tell <- _TELL_telling_4 $ url , "Hear.zxe": eVal, sVal;

// -----

imbeddedType ibt;
ibt.e = _SIMPLE_enum_3;
ibt.s = "imbeddedType, reset by main...";

tell <- _TELL_telling_5 $ url , "Hear": ibt;

// -----

output << "Teller: Good-bye World!\n";

#endif // HearModeOn

end;
```