Script reference

ezeio script programming - the PAWN language

The scripting feature in the ezeio allows the user to implement advanced custom logic and control functionality on the ezeio. For most common applications, scripting is not necessary.

The ezeio uses a powerful script language called PAWN. The syntax is very similar to JavaScript, and should be easily understood by anyone working with programming in similar languages.

The scripting feature of the ezeio is intended for those with previous experience in programming. This manual does not attempt to teach you to write programs. If you have a specific feature that you need help with, please contact eze System.

We strongly recommend reading the following documents to get introduced to the PAWN language:

Introduction to PAWN

PAWN language guide

ezeio script administration

When working with ezeio scripts, you will edit your code on the userscript screen. When clicking “Commit”, the script is compiled on the servers, and automatically downloaded into the ezeio, where it will start to run as soon as the download completes.

Only one user defined script can run at the same time, but the editor allows you to have multiple scripts saved on the servers, and easily switch between them. The currently running script is marked with a checkmark.

If a script fails to compile due to a syntax error, the script will not be sent to the ezeio. If a previous version of the script, or a different script, was running, this will continue to run.

Script resources

The user script can occupy up to 128kB code (compiled bytecode), and up to 16kB RAM.

The estimated requirements for a script is displayed when compiling.

Programming pattern

Although the user script runs in a sandboxed runtime engine, the recommended programming pattern is similar to cooperative multitasking.

This means that you should avoid long-running loops, and instead make use of system callbacks provided in the function library.

The following system callback functions are defined:

@Tick	Called at a regular interval, set by SetTickInterval()
@Timer	Called when one of the millisecond timers expire, see SetTimer()
@Action	Called when an action is triggered due to an alarm event
@Key	Called when a button is pressed on a connected terminal device
@Scan	Called when a code is received from a scanner device

Some examples

The program below will print out “Hello” on the debug console.

```
main()
{
    PDebug("Hello");
}
```

The following example adds the values of field 1 and field 2 together, and writes the result to field 3.

```
// This example adds the value of field 1 and 2 and writes the sum to field 3
// The value of field 3 is updated every 500ms (twice per second)

main()
{
    SetTickInterval(500);       // Make sure the @Tick function is called
twice per second
}

@Tick(uptime)
{
    new f1, f2;             // declare local variables

    f1 = GetField(1);       // set f1 to the value of field 1
    f2 = GetField(2);       // set f2 to the value of field 2

    SetField(3, f1+f2);     // set the value of field 3 with the sum of f1 and
f2
}
```

The example below illustrates the use of a global variable, the use of the main() function to initialize the values, and the @Tick(uptime) call. The program will use field number 1, and count from 100 down to 50, and then starting over.

```
// A simple demo program, counting down from 100 to 50, and then starts over

new myCounter;                  // Declare a 'global' variable

main()                          // The main() function will run on startup,
once
{
    SetTickInterval(1000);      // Make sure the @Tick function is called once
every 1000ms (1s)
    myCounter = 100;            // Initialize the counter with value 100
}

@Tick(uptime)                   // The @Tick function will be called once per
second
{
    myCounter = myCounter - 1;  // Count down the counter with 1

    if(myCounter <= 50) {       // If we reached 50 (or lower)
        myCounter = 100         // ..then set it back to 100
    }

    SetField(1, myCounter);     // Update field 1 with the counter value
}
```

The example below will run a 3-speed fan based on the input from a temperature sensor on field 1. Output 1 is used for low speed, output 2 is medium speed, and output 3 is high speed. The speed is selected based on how much the temperature exceeds a given setpoint (here set to 20.5 degrees C)

```
main()
{
    SetTickInterval(5000);  // Update every 5s (5000ms)
}

@Tick(uptime)
{
    new Float:temp, Float:diff; // Declare "float" variables to handle
fractions

    temp = GetFieldFloat(1);    // Fetch the current temperature from Field 1
    diff = temp - 20.5;         // 20.5 is our setpoint. diff is the differece

    if(diff > 0.0)              // Over setpoint?
        SetOutput(1, 100);      // ..yes, so enable low speed
    else
        SetOutput(1, 0);        // ..no, so shut off

    if(diff > 2.0)              // 2 degrees or more over setpoint?
```

```
        SetOutput(2, 100);      //..yes, so enable mid-speed
    else
        SetOutput(2, 0);

    if(diff > 6.0)              // 6 degrees or more over setpoint?
        SetOutput(3, 100);      //..yes, so run full speed
    else
        SetOutput(3, 0);
}
```

State machines

A common programming pattern in control applications is to use state machines. PAWN and the ezeio implements strong support for state machines. The following is a typical pattern showing the startup sequence of an engine. Note that there are three @Tick handlers; one for each state. Also note the “entry” and “exit” functions. For more detail, refer to the PAWN language guide.

```
  new count = 0;

  main()
  {
    SetTickInterval(100);   // set tick interval to 100ms (0.1s)
    state WAITING;          // start in the waiting mode
  }

  @Tick(uptime) <WAITING>
  {
     if( GetField(1) < 100 )  // condition to start up the process
        state IGNITION;
  }

// ***** the IGNITION state

  entry() <IGNITION>
  {
     SetOutput(1, 100);       // Turn on the ignition switch
     count = 0;
  }

  @Tick(uptime) <IGNITION>
  {
     if( GetField(2) > 100 )  // Did the engine start?
        state RUNNING;        // yes - we're running

     if(count++ > 50)
        state WAITING;        // didn't start in 5s? Give up and go back to
```

```
waiting.
  }

  exit() <IGNITION>
  {
     SetOutput(1, 0);          // Turn the ignition switch off
  }

// ***** the RUNNING state

  @Tick(uptime) <RUNNING>
  {
     if( GetField(2) < 100 )  // Check if the engine stopped
        state WAITING;        // ..go back to waiting
  }
```