xpspl Documentation

Release 5.0.0

Nickolas Whiting

Apr 17, 2017

Contents

1 Table of Contents 3

2 Modules 37

3 Source 45

4 Support 47

5 Author 49

XPSPL is a high-performance event loop for PHP that supports the following event types.

- Signals
- Timers
- CRON Timers
- Asynchronous Network I/O
- Complex Signals
- Idle

The best way to think of XPSPL is as a libevent and libev library only written in PHP.

CHAPTER 1

Table of Contents

API

XPSPL interfaces a globally available processor using API functions.

The API functions are namespaced with *xp* prepended to all functions.

xp_after

xp_after(*$signal*, *$process*)

Execute a function after a signal has been emitted.

Parameters

- **object** – \XPSPL\SIG
- **callable|process** – PHP Callable or \XPSPL\Process.

Return type object \XPSPL\Process

Example #1 Basic Usage

```
<?php

xp_signal(XP_SIG('foo'), function(){
    echo 'foo';
});

xp_after(XP_SIG('foo'), function(){
    echo 'after foo';
});
```

The above code will output.

```
// fooafter foo
```

xp_before

xp_before (*$signal*, *$process*)

Execute a function before a signal is emitted.

Parameters

- **object** – \XPSPL\SIG
- **callable|process** – PHP Callable or \XPSPL\Process.

Return type object \XPSPL\Process

Example #1 Basic Usage

```
<?php

xp_signal(XP_SIG('foo'), function(){
    echo 'foo';
});

xp_before(XP_SIG('foo'), function(){
    echo 'before foo';
});

// results when foo is emitted
```

The above code will output.

```
// before foo foo
```

xp_clean

xp_clean ([*$history* = *false*])

Scans and removes non-emittable signals and processes.

Note: This DOES NOT flush the processor.

A signal is determined to be emittable only if it has installed processes that have not exhausted.

Parameters **boolean** – Erase any history of removed signals.

Return type void

Example #1 Basic Usage

Basic usage example demonstrating cleaning old signals and processes.

```php
<?php

xp_signal(XP_SIG('Test'), xp_exhaust(1, function(){
    echo 'SIG Test';
}));

xp_signal(XP_SIG('Test_2'), function(){
    echo 'SIG Test 2';
});

xp_emit(XP_SIG('Test'));

xp_clean();
var_dump(xp_find_signal(XP_SIG('Test')));
```

The above code will output.

```
SIG Test null
```

xp_complex_sig

xp_complex_sig (*$function*[, *$rebind_context = true*])

Allows for performing complex signal processing using callable PHP variables.

A `\Closure` will be bound into an object context which allows maintaining variables across emits within `$this`.

Note: A `\Closure` can remain bound to its original context by passing `$rebind_context` as `false`.

Parameters

- **callable** – Callable variable to use for evaluation.
- **boolean** – Rebind the given closures context to this object.

Return type \XPSPL\SIG_Complex Complex signal registered to the processor.

Example #1 Detecting a wedding.

```php
<?php
// Once a bride, groom and bell signals are emitted we emit the wedding.
$wedding = xp_complex_sig(function($signal){
    if (!isset($this->reset) || $this->reset) {
        $this->reset = false;
        $this->bride = false;
        $this->groom = false;
        $this->bells = false;
    }
    switch (true) {
        // Signals can be compared using the compare method
        // this will return if the signals are identical
        case $signal->compare(XP_SIG('groom')):
            $this->groom = true;
            break;
```

```
        case $signal->compare(XP_SIG('bride')):
            $this->bride = true;
            break;
        case $signal->compare(XP_SIG('bells')):
            $this->bells = true;
            break;
    }
    if ($this->groom && $this->bride && $this->bells) {
        $this->reset = true;
        return true;
    }
    return false;
});

xp_signal($wedding, function(){
    echo 'A wedding just happened.';
});

xp_emit(XP_SIG('bride'));
xp_emit(XP_SIG('groom'));
xp_emit(XP_SIG('bells'));
```

The above code will output.

```
A wedding just happened.
```

Example #2 Detecting a wedding using network recieved signals.

```
<?php

xp_import('network');

$server = network\connect('0.0.0.0', ['port' => 8000]);
// Setup a server that emits a signal of recieved data
$server->on_read(function($signal){
    $read = trim($signal->socket->read());
    if ($read == null) {
      return false;
    }
    xp_emit(XP_SIG($read));
});

// Once a bride, groom and bell signals are emitted we emit the wedding.
$wedding = xp_complex_sig(function($signal){
    if (!isset($this->reset) || $this->reset) {
      $this->reset = false;
      $this->bride = false;
      $this->groom = false;
      $this->bells = false;
    }
    switch (true) {
      case $signal->compare(XP_SIG('groom')):
          $this->groom = true;
          break;
      case $signal->compare(XP_SIG('bride')):
          $this->bride = true;
```

```
            break;
        case $signal->compare(XP_SIG('bells')):
            $this->bells = true;
            break;
    }
    if ($this->groom && $this->bride && $this->bells) {
      $this->reset = true;
      return true;
    }
    return false;
});

xp_signal($wedding, function(){
    echo 'A wedding just happened.';
});

// Start the wait loop
xp_wait_loop();
```

The above code will output.

```
A wedding just happened.
```

Once the `bride`, `groom` and `bells` signals are emitted from the network connection the complex signal will emit the wedding.

xp_current_signal

xp_current_signal ([*$offset = false*])
Retrieve the current signal in execution.

Parameters **integer** – If a positive offset is given it will return from the top of the signal stack, if negative it will return from the bottom (current) of the stack.

Return type object \XPSPL\SIG

Example #1 Basic Usage

```
<?php

xp_signal(XP_SIG('foo'), function(\XPSPL\SIG $signal){
    $a = xp_current_signal();
    echo $a->get_index();
});
```

The above code will output.

```
foo
```

Example #2 Parent Signals

Parent signals can be fetched by using a negative offset `< -1`.

```
<?php

// Install a process on the bar SIG
xp_signal(XP_SIG('bar'), function(){
    // Emit foo within bar
    xp_emit(XP_SIG('foo'));
});

// Install a process on the foo SIG
xp_signal(XP_SIG('foo'), function(){
    // Get the parent of foo
    $a = xp_current_signal(-2);
    echo $a->get_index();
});
```

The above code will output.

```
bar
```

xp_delete_process

xp_delete_process (*$signal*, *$process*)

Deletes an installed signal process.

> **Note:** The \XPSPL\Process object given must be the same returned or created when the process was installed.

Parameters

- **object** – \XPSPL\SIG signal to remove process from.
- **object** – \XPSPL\Process object to remove.

Return type void

Example #1 Basic Usage

The above code will output.

```
bar
```

xp_delete_signal

xp_delete_signal (*$signal*[, *$history = false*])

Delete a signal from the processor.

Parameters

- **string|object|int** – Signal to delete.
- **boolean** – Erase any history of the signal.

Return type boolean

Example #1 Basic Usage

```
<?php
// Install process on signal foo
xp_signal(XP_SIG('foo'), function(){});
// Delete the signal foo
xp_delete_signal(XP_SIG('foo'));
// Emit the signal foo
xp_emit(XP_SIG('foo'));
```

xp_dir_include

xp_dir_include (*$dir*[, *$listen = false*[, *$path = false*]])

Recursively includes all .php files in the given directory.

Listeners can be started automatically by passing `$listen` as `true`.

param string Directory to include.

param boolean Start listeners.

param string Path to ignore when starting listeners.

rtype void .. note:

Listener class names are generated compliant to PSR-4 with the directory serving as the top-level namespace.

Example #1 Basic Usage

```
xp_dir_include('Foo');
```

With the directory structure.

```
- Foo/
    - Bar.php
```

Will include the file Foo/Bar.php

Example #2 Listeners

```
xp_include_dir('Foo', true);
```

With the directory structure.

```
- Foo/
    - Bar.php
    - Bar/
        - Hello_World.php
```

Will include the files `Foo/Bar.php`, `Foo/Bar/Hello_World.php` and attempt to start classes `Foo\Bar`, `Foo\Bar\Hello_World`.

Note: Listeners must extend the XPSPL\Listener class.

```php
<?php
namespace Foo\Bar;

Class Hello_World extends \XPSPL\Listener {

    // Do something on the 'foo' signal
    public function on_foo(\XPSPL\SIG $signal) {
        echo 'foobar';
    }

}
```

When the `XP_SIG('foo')` signal is emitted the `Hello_World->on_foo` method will be executed.

xp_emit

xp_emit (*$signal*[, *$context = false*])

Emits a signal.

This will execute all processes, before and after functions installed to the signal.

Returns the executed `\XPSPL\SIG` object.

Note: When emitting unique signals, e.g.. complex, routines or defined uniques the unique `\XPSPL\SIG` object installed must be provided.

Once a signal is emitted the following execution chain takes place.

1. Before process functions
2. Process function
3. After process function

Parameters

- **signal** – XPSPLSIG object to emit.
- **object** – XPSPLSIG object context to execute within.

Return type object XPSPLSIG

Example #1 Basic Usage

```php
<?php

// Install a process on the foo signal
xp_signal(XP_SIG('foo'), function(){
    echo 'foo';
});

// Emit the foo signal
xp_emit(XP_SIG('foo'));
```

The above example will output.

```
foo
```

Example #2 Unique Signals

```php
<?php

// Create a unique Foo signal.
class SIG_Foo extends \XPSPL\SIG {
    // declare it as unique
    protected $_unique = true;
}

// Create a SIG_Foo unique object
$foo = new SIG_Foo();

signal($foo, function(){
    echo "foo";
});

// Emit the SIG_Foo and another unique SIG_Foo
xp_emit($foo);
xp_emit(new Foo());
```

The above code will output.

```
foo
```

Example #3 Complex Signals

```php
<?php

// Create our 3 required signals.
class SIG_Bells extends \XPSPL\SIG {}
class SIG_Bride extends \XPSPL\SIG {}
class SIG_Groom extends \XPSPL\SIG {}

// Create a complex signal that will emit when a wedding takes place
// based on 3 seperate signals, SIG_Bells, SIG_Bride and SIG_Groom.
class SIG_Wedding extends \XPSPL\SIG_Complex {

    // Keep track if each requirement has already emitted
    protected $_bells = false;
    protected $_bride = false;
    protected $_groom = false;

    // Create an evaulation method to evaluate the runtime
    public function evaluate($signal = null)
    {
      switch ($signal) {
        case $signal instanceof SIG_Bells:
            $this->_bells = true;
            break;
        case $signal instanceof SIG_Bride:
            $this->_bride = true;
```

```
            break;
        case $signal instanceof SIG_Groom:
            $this->_groom = true;
            break;
      }
      if ($this->_bells === true &&
            $this->_bride === true &&
            $this->_groom === true) {
        $this->_bells = false;
        $this->_groom = false;
        $this->_bride = false;
        return true;
      }
      return false;
    }
}

// Install a process for complex signal.
xp_signal(new SIG_Wedding(), function(){
    echo 'The wedding is happening!'.PHP_EOL;
});

// Emit SIG_Bells, SIG_Bride and SIG_Groom
xp_emit(new SIG_Bells());
xp_emit(new SIG_Bride());
xp_emit(new SIG_Groom());
```

The above code will output.

```
The wedding is happening!
```

xp_erase_history

xp_erase_history()
Erases the entire signal history.

Return type void

Example #1 Basic Usage

```
<?php

// Create some history
xp_signal(XP_SIG('foo'), function(){});
for ($i=0;$i<10;$i++) {
    xp_emit(XP_SIG('foo'));
}

// Dump the history count
var_dump(count(xp_signal_history()));

// Erase the history
xp_erase_history();

var_dump(count(xp_signal_history()));
```

The above code will output.

```
10 ... 0
```

xp_erase_signal_history

xp_erase_signal_history(*$signal*)
Erases the history of only the given signal.

> **Warning:** This will delete the history for *ANY* signals that are a direct child of the to be deleted signal.
>
> As an example,
>
> When `sig_foo` emits it is proceeded directly by `sig_foo2` emitting within the `sig_foo` execution.
>
> When sig_foo is deleted the history of sig_foo_child will also be removed.

Parameters **string|object** – Signal to be erased from history.

Return type void

Example #1 Basic Usage

```
<?php

// Install a procss for the foo and bar signals.
xp_signal(XP_SIG('foo'), function(){});
xp_signal(XP_SIG('bar'), function(){});
// Emit each foo and bar 10 times.
for($i=0;$i<10;$i++) {
    xp_emit(XP_SIG('foo'));
    xp_emit(XP_SIG('bar'));
}
var_dump(count(xp_signal_history()));
// Delete the history of the foo signal.
xp_delete_signal_history(XP_SIG('foo'));
var_dump(count(xp_signal_history()));
```

The above code will output.

```
20 ... 10
```

xp_exhaust

xp_exhaust(*$limit*, *$process*)
Defines the number of times a process will execute when a signal is emitted.

> **Note:** By default all processes have an exhaust rate of null.

Parameters **callable|process** – PHP Callable or Process.

Return type object Process

Example #1 Basic Usage

Defines the given process with an exhaust of 5.

```
<?php

       // Install a process for the foo signal that will execute up to 5 times.
xp_signal(XP_SIG('foo'), xp_exhaust(5, function(){
    echo 'foo';
});

for($i=0;$i<10;$i++){
    xp_emit('foo');
}
```

The above code will output.

```
foofoofoofoofoo
```

Example #2 Creating a timeout

```
<?php

// Import the time module
xp_import('time');

time\awake(10, xp_exhaust(1, function(){
    echo 'This will execute only once.';
});
```

The above code will output.

```
This will execute only once.
```

xp_high_priority

xp_high_priority(*$process*)

Creates or sets a process to have a high priority.

Processes with a high priority will be executed before those with a low or default priority.

This will register the priority as *0* as priority goes in ascending order.

Note: Interruptions will be executed before high priority processes.

Parameters **callable|process** – PHP Callable or XPSPLProcess.

Return type object Process

Example #1 Basic Usage

Basic usage example demonstrating high priority processes.

```php
<?php

// Register a process on the foo signal
xp_signal(XP_SIG('foo'), function(){
    echo 'bar';
});

// Register another process with high priority
xp_signal(XP_SIG('foo'), xp_high_priority(function(){
    echo 'foo';
}));
```

The above code will output.

```
foobar
```

xp_import

xp_import (*$name*[, *$dir = false*])

Import a module for usage.

By default modules will be loaded from the `modules/` directory located within XPSPL.

Parameters

- **string** – Module name.
- **string|null** – Location of the module.

Return type void

Example #1 Basic Usage

```php
<?php

// Import the time module
xp_import('time');
```

Example #2 Importing modules from user-defined directories

```php
<?php

// Import the "foobar" module from our custom module directory
xp_import('foobar', '/my-custom/directory/path');
```

xp_listen

xp_listen (*$listener*)

Registers a new listener.

Listeners are special objects that register each publically available method as an executing process using the method name.

Note: Public methods that are declared with a prepended underscore "_" are ignored.

Parameters **object** – The object to register.

Return type void

Example #1 Basic Usage

```
<?php

class My_Listener extends \XPSPL\Listener
{
    // Register a process for the foo signal.
    public function foobar($signal) {
        echo 'foobar';
    }
}

xp_listener(new My_Listener());

xp_emit(XP_SIG('foobar'));
```

The above code will output.

```
foobar
```

xp_low_priority

xp_low_priority(*$process*)

Creates or sets a process to have a low priority.

Processes with a low priority will be executed after those with a high priority.

Note: This registers the priority as *PHP_INT_MAX*.

This is not an interruption.

After signal interrupts will still be executed after a low priority process.

Parameters **callable|process** – PHP Callable or XPSPLProcess.

Return type object Process

Example #1 Basic Usage

Low priority processes always execute last.

```
<?php

xp_signal(XP_SIG('foo'), xp_low_priority(function(){
    echo 'bar';
}));
```

```
xp_signal(XP_SIG('foo'), function(){
    echo 'foo';
});

xp_emit(XP_SIG('foo'));
```

The above code will output.

```
foobar
```

xp_null_exhaust

xp_null_exhaust (*$process*)

Nullifies a processes exhaustion rate.

Note: Once a process is registered with a null exhaust it will **never** be purged from the processor unless a `xp_flush` is called.

Parameters **callable|process** – PHP Callable or Process.

Return type object Process

Example #1 Basic Usage

This example installs a null exhaust process which calls an awake signal every 10 seconds creating an interval.

```
<?php
import('time');

time\awake(10, xp_null_exhaust(function(){
    echo "10 seconds";
}));
```

xp_on_shutdown

xp_on_shutdown (*$function*)

Registers a function to call when the processor shuts down.

Parameters **callable|object** – Function or process object

Return type object XPSPLProcess

Example #1 Basic Usage

```
<?php

xp_on_shutdown(function(){
    echo 'Shutting down the processor!';
});
```

```
xp_wait_loop();
```

The above code will output.

```
Shutting down the processor!
```

xp_on_start

xp_on_start (*$function*)

Registers a function to call when the processor starts.

Parameters **callable|object** – Function or process object

Return type object XPSPLProcess

Example #1 Basic Usage

```
<?php

xp_on_start(function(){
    echo 'The processor started';
});

xp_wait_loop();
```

The above code will output.

```
The processor started!
```

xp_priority

xp_priority (*$priority*, *$process*)

Sets the priority of a process.

This allows for controlling the order of processes rather than using FIFO.

Priority uses an ascending order where 0 > 1.

Processes registered with a high priority will be executed before those with a low or default priority.

Process priority is handy when multiple process will execute and their order is important.

Note: This is different from an interrupt.

Installed interrupts will still be executed before or after a prioritized process.

Parameters

- **integer** – Priority to assign
- **callable|process** – PHP Callable or XPSPLProcess.

Return type object Process

Example #1 Basic Usage

This installs multiple process each with a seperate ascending priority.

```
<?php

xp_signal(XP_SIG('foo'), priority(0, function(){
    echo 'foo';
}));

xp_signal(XP_SIG('foo'), priority(3, function(){
    echo 'bar';
}));

xp_signal(XP_SIG('foo'), priority(5, function(){
    echo 'hello';
}));

xp_signal(XP_SIG('foo'), priority(10, function(){
    echo 'world';
}));
```

The above code will output.

```
foobarhelloworld
```

xp_process

xp_process (*$callable*)
Generates a XPSPLProcess object from the given PHP callable.

> **Note:** See the `priority` and `exhaust` functions for setting the priority and exhaust of the created process.

Parameters **callable** –

Return type void

Example #1 Basic Usage

Creates a new XPSPL Process object.

xp_register_signal

xp_register_signal (*$signal*)
Registers a signal in the processor.

Parameters **string|integer|object** – Signal

Return type object Database

xp_set_signal_history

xp_set_signal_history(*$flag*)
Sets the flag for storing the event history.

Parameters **boolean** –

Return type void

xp_shutdown

xp_shutdown()
Sends the event loop the shutdown signal.

Return type void

XP_SIG

XP_SIG(*$signal*)
Generates an XPSPL SIG object from the given `$signal`.

This function is only a shorthand for `new XPSPL\SIG($signal)`.

Parameters **string|** – Signal process is attached to.

Return type object XPSPLSIG

Example #1 Basic Usage

This will create a SIG idenitified by 'foo'.

```
<?php
xp_signal(XP_SIG('foo'), function(){
    echo "HelloWorld";
});

xp_emit(XP_SIG('foo'));
```

xp_signal

xp_signal(*$signal*, *$process*)
Registers a process to execute when the given signal is emitted.

Note: All processes by default have an exhaust of `null`.

Note: Processes installed to the same signal execute in FIFO order when no priority is defined.

Parameters

- **object** – Signal to install process on.
- **object** – PHP Callable

Return type object | boolean - XPSPLProcess otherwise boolean on error

---

Note: Beginning in XPSPL v4.0.0 all signals were converted to strictly objects.

To use a string or integer as a signal it must be wrapped in a `XP_SIG`.

Warning: Any signal created using `` `XP_SIG` `` CANNOT be unique.

Example #1 Basic Usage

```
<?php

// Install a process on the foo signal
xp_signal(XP_SIG('foo'), function(){
    echo 'foo';
});

// Emit the foo signal
xp_emit(XP_SIG('foo'));
```

The above example will output.

```
foo
```

Example #2 Class Signals

```
<?php

// Declare a simple signal
class SIG_Basic extends \XPSPL\SIG {}

// Install a process on the SIG_Basic class
xp_signal(new SIG_Basic(), function(){
    echo 'foo';
})

xp_emit(new SIG_Basic());
```

The above example will output.

```
foo
```

Example #3 Exhausting Processes

```
<?php

// Install a process on the foo signal, with an exhaust of 1
xp_signal(XP_SIG('foo', xp_exhaust(1, function(){
    echo 'foo';
```

```
}));

// Emit the foo signal
xp_emit(XP_SIG('foo'));
xp_emit(XP_SIG('foo'));
```

The above code will output.

```
foo
```

Example #4 Unique Signals

```
<?php

// Declare a simple unique
class SIG_Foo extends \XPSPL\SIG {
    // Set the signal as unique
    protected $_unqiue = true;
}

// Create two unique SIG_Foo objects
$sig_foo_1 = new SIG_Foo();
$sig_foo_2 = new SIG_Foo();

// Install a process to each unique signal
xp_signal($sig_foo_1, function(){
    echo 'foo';
});

xp_signal($sig_foo_2, function(){
    echo 'bar';
})

// Emit each unique signal
xp_emit($sig_foo_1);
xp_emit($sig_foo_2);
```

The above code will output.

```
foobar
```

xp_signal_history

xp_signal_history()
Returns the current signal history.

The returned history is stored in an array using the following indexes.

```
<?php
$array = [
    0 => Signal Object
    1 => UNIX timestamp of execution
];
```

Return type array

Example #1 Basic Usage

Count the number of `XP_SIG('foo')` signals that were emitted.

```
<?php
$sig = XP_SIG('foo');
// Emit a few foo objects
for($i=0;$i<5;$i++){
    xp_emit($sig);
}
$emitted = 0;
foreach(xp_signal_history() as $_node) {
    if ($_node[0] instanceof $sig) {
        $emitted++;
    }
}
echo $emitted;
```

The above code will output.

```
5
```

xp_threaded_process

xp_threaded_process(*$process*)

Enables a process to execute within it's own thread.

> **Warning:** Threaded functionality within XPSPL is *highly* experiemental...
>
> This has not been tested in a production environment.

Note: To enable threads you must install and enable the PECL pthreads extension.

Once installed threads will be automatically enabled.

Example #1 Basic Usage

Executing processes in their own thread.

```
<?php

xp_signal(XP_SIG('foo'), threaded_process(function($sig){
    print 'Executed in own thread';
    sleep(10);
});

 :param callable: PHP Callable

 :rtype: void
```

xp_wait_loop

xp_wait_loop()
Begins the XPSPL event wait loop.

The event loop must be started to allow execution of time, networking or complex loop based signals.

Note: XPSPL provides an executable `xpspl` in the *bin* directory for automatically loading code into the event loop.

> **Warning:** This is a *BLOCKING* function.
>
> Any code underneath the function call will *NOT* be executed until the processor halts execution.

Return type void

Example #1 Basic Usage

Basic usage example demonstrating using the loop for time based code.

```
<?php

// Import time module
xp_import('time');

xp_time\awake(10, function(){
    echo '10 seconds passed';
});

xp_wait_loop();
```

Automatic shutdown

The processor loop has built in support for automatically shutting down when it detects there is nothing else it will ever do.

This example demonstrates the loop shutting down after emitting 5 time based signals.

```

```

xp_get_signal

xp_get_signal(*$signal*)
Returns an installed signal database or null if it cannot be found.

Parameters **object** – SIG

Return type nullobject XPSPLdatabaseSignals

XPSPL

XPSPL()
Returns the global XPSPL processor.

Return type object XPSPLProcessor

XPSPL_flush

XPSPL_flush()
Performs a complete flush of the processor.

This will clear the processor state and remove the following.

- Installed signals.
- Installed processes.
- Signal history.

Return type void

Configuration

Configuration values are defined using constants before loading XPSPL.

Constants

XPSPL_DEBUG

XPSPL Debug mode

When turned on XPSPL generates a log of all activity to STDOUT.

When turned off XPSPL removes its processor traces from uncaught exceptions.

XPSPL_SIGNAL_HISTORY

Signal History

Default setting for the saving the signal history.

By default this is `false`.

XPSPL_PURGE_EXHAUSTED

Remove Exhausted processes

When turned on this automatically removes installed processes from the processor once it determines they can no longer be used.

By default this settings is `true`.

XPSPL_MODULE_DIR

Module Directory

Directory to look for modules.

By default it is set to the `module` directory in XPSPL.

XPSPL_PROCESS_DEFAULT_EXHAUST

Default process exhaustion

Integer option defining the default exhausting of a process.

By default it is `1`.

XPSPL_PROCESS_DEFAULT_PRIORITY

Process default priority

Integer option defining the default priority of all processes.

By default it is `10`.

XPSPL_JUDY_SUPPORT

Judy is an optional database configuration.

http://xpspl.prggmr.org/en/xspel/install.html#optional

Currently this is experimental as an attempt to improve performance.

Once stable this will automatically be enabled if Judy is detected.

XPSPL_ANALYZE_TIME

UNUSED

This is an unused configuration option that will later add support for analyzing the processor timing to auto correct signal timing ... at least that is the theory.

Last updated on 04/23/13 11:50pm

Install

Requirements

XPSPL requires **>= PHP 5.4**.

Composer

XPSPL is installed using (composer).

```
require: "prggmr/xpspl": "v5.0.0"
```

Once installed XPSPL's API will be available after including composer's `vender/autoload.php`.

Manual

> **Warning:** This is the legacy installation method.
>
> Only install XPSPL using this if you are using XPSPL < v5.0.0

The manual installation is over the network with a CLI. (link)

The installation requires the **CURL** and **ZIP** libraries to be installed on the system and is the legacy installation.

```
curl -s https://raw.github.com/prggmr/xpspl/master/install | sudo sh
```

Optional

FTP Module

PHP FTP extension for XPSPL FTP module support.

Networking

PHP Sockets extension for XPSPL Networking module support.

Judy

C Judy 1.0.4 PECL Judy 0.1.4

The Judy library demonstrates improving the database by giving sptorage a linear average performance of 39us per write to a tested 262144.

For installation of Judy C see the README.

For installation of Judy PECL visit here.

Quickstart

Contents

- *Quickstart*
  - *Installing XPSPL*

- *Processing and Emitting signals*
  - *xp_signal*
    - *Example #1 Basic Usage*
    - *Example #2 Class Signals*
    - *Example #3 Exhausting Processes*
    - *Example #4 Unique Signals*
  - *xp_emit*
    - *Example #1 Basic Usage*
    - *Example #2 Unique Signals*
    - *Example #3 Complex Signals*
  - *xp_complex_sig*
    - *Example #1 Detecting a wedding.*
    - *Example #2 Detecting a wedding using network recieved signals.*
- *Environment*

Installing XPSPL

XPSPL is installed using (composer).

```
require: "prggmr/xpspl": "v5.0.0"
```

Once installed XPSPL's API will be available after including composer's `vender/autoload.php`.

Processing and Emitting signals

Processing and emitting signals is performed using the `xp_signal`, `xp_emit` and `xp_complex_sig` functions.

xp_signal

xp_signal (*$signal*, *$process*)

Registers a process to execute when the given signal is emitted.

Note: All processes by default have an exhaust of `null`.

Note: Processes installed to the same signal execute in FIFO order when no priority is defined.

Parameters

- **object** – Signal to install process on.
- **object** – PHP Callable

Return type object | boolean - XPSPLProcess otherwise boolean on error

Note: Beginning in XPSPL v4.0.0 all signals were converted to strictly objects.

To use a string or integer as a signal it must be wrapped in a `XP_SIG`.

Warning: Any signal created using `` `XP_SIG` `` CANNOT be unique.

Example #1 Basic Usage

```php
<?php

// Install a process on the foo signal
xp_signal(XP_SIG('foo'), function(){
    echo 'foo';
});

// Emit the foo signal
xp_emit(XP_SIG('foo'));
```

The above example will output.

```
foo
```

Example #2 Class Signals

```php
<?php

// Declare a simple signal
class SIG_Basic extends \XPSPL\SIG {}

// Install a process on the SIG_Basic class
xp_signal(new SIG_Basic(), function(){
    echo 'foo';
})

xp_emit(new SIG_Basic());
```

The above example will output.

```
foo
```

Example #3 Exhausting Processes

```php
<?php

// Install a process on the foo signal, with an exhaust of 1
xp_signal(XP_SIG('foo', xp_exhaust(1, function(){
    echo 'foo';
})));
```

```
// Emit the foo signal
xp_emit(XP_SIG('foo'));
xp_emit(XP_SIG('foo'));
```

The above code will output.

```
foo
```

Example #4 Unique Signals

```
<?php

// Declare a simple unique
class SIG_Foo extends \XPSPL\SIG {
    // Set the signal as unique
    protected $_unqiue = true;
}

// Create two unique SIG_Foo objects
$sig_foo_1 = new SIG_Foo();
$sig_foo_2 = new SIG_Foo();

// Install a process to each unique signal
xp_signal($sig_foo_1, function(){
    echo 'foo';
});

xp_signal($sig_foo_2, function(){
    echo 'bar';
})

// Emit each unique signal
xp_emit($sig_foo_1);
xp_emit($sig_foo_2);
```

The above code will output.

```
foobar
```

xp_emit

xp_emit (*$signal*[, *$context = false*])

Emits a signal.

This will execute all processes, before and after functions installed to the signal.

Returns the executed `\XPSPL\SIG` object.

Note: When emitting unique signals, e.g.. complex, routines or defined uniques the unique `\XPSPL\SIG` object installed must be provided.

Once a signal is emitted the following execution chain takes place.

1.Before process functions

2.Process function

3.After process function

Parameters

- **signal** – XPSPLSIG object to emit.
- **object** – XPSPLSIG object context to execute within.

Return type object XPSPLSIG

Example #1 Basic Usage

```
<?php

// Install a process on the foo signal
xp_signal(XP_SIG('foo'), function(){
    echo 'foo';
});

// Emit the foo signal
xp_emit(XP_SIG('foo'));
```

The above example will output.

```
foo
```

Example #2 Unique Signals

```
<?php

// Create a unique Foo signal.
class SIG_Foo extends \XPSPL\SIG {
    // declare it as unique
    protected $_unique = true;
}

// Create a SIG_Foo unique object
$foo = new SIG_Foo();

signal($foo, function(){
    echo "foo";
});

// Emit the SIG_Foo and another unique SIG_Foo
xp_emit($foo);
xp_emit(new Foo());
```

The above code will output.

```
foo
```

Example #3 Complex Signals

```php
<?php

// Create our 3 required signals.
class SIG_Bells extends \XPSPL\SIG {}
class SIG_Bride extends \XPSPL\SIG {}
class SIG_Groom extends \XPSPL\SIG {}

// Create a complex signal that will emit when a wedding takes place
// based on 3 seperate signals, SIG_Bells, SIG_Bride and SIG_Groom.
class SIG_Wedding extends \XPSPL\SIG_Complex {

    // Keep track if each requirement has already emitted
    protected $_bells = false;
    protected $_bride = false;
    protected $_groom = false;

    // Create an evaulation method to evaluate the runtime
    public function evaluate($signal = null)
    {
      switch ($signal) {
        case $signal instanceof SIG_Bells:
            $this->_bells = true;
            break;
        case $signal instanceof SIG_Bride:
            $this->_bride = true;
            break;
        case $signal instanceof SIG_Groom:
            $this->_groom = true;
            break;
      }
      if ($this->_bells === true &&
            $this->_bride === true &&
            $this->_groom === true) {
        $this->_bells = false;
        $this->_groom = false;
        $this->_bride = false;
        return true;
      }
      return false;
    }
}

// Install a process for complex signal.
xp_signal(new SIG_Wedding(), function(){
    echo 'The wedding is happening!'.PHP_EOL;
});

// Emit SIG_Bells, SIG_Bride and SIG_Groom
xp_emit(new SIG_Bells());
xp_emit(new SIG_Bride());
xp_emit(new SIG_Groom());
```

The above code will output.

```
The wedding is happening!
```

xp_complex_sig

xp_complex_sig (*$function*[, *$rebind_context* = *true*])

Allows for performing complex signal processing using callable PHP variables.

A `\Closure` will be bound into an object context which allows maintaining variables across emits within `$this`.

Note: A `\Closure` can remain bound to its original context by passing `$rebind_context` as `false`.

Parameters

- **callable** – Callable variable to use for evaluation.
- **boolean** – Rebind the given closures context to this object.

Return type \XPSPL\SIG_Complex Complex signal registered to the processor.

Example #1 Detecting a wedding.

```
<?php
// Once a bride, groom and bell signals are emitted we emit the wedding.
$wedding = xp_complex_sig(function($signal){
    if (!isset($this->reset) || $this->reset) {
        $this->reset = false;
        $this->bride = false;
        $this->groom = false;
        $this->bells = false;
    }
    switch (true) {
        // Signals can be compared using the compare method
        // this will return if the signals are identical
        case $signal->compare(XP_SIG('groom')):
            $this->groom = true;
            break;
        case $signal->compare(XP_SIG('bride')):
            $this->bride = true;
            break;
        case $signal->compare(XP_SIG('bells')):
            $this->bells = true;
            break;
    }
    if ($this->groom && $this->bride && $this->bells) {
        $this->reset = true;
        return true;
    }
    return false;
});

xp_signal($wedding, function(){
    echo 'A wedding just happened.';
});

xp_emit(XP_SIG('bride'));
xp_emit(XP_SIG('groom'));
xp_emit(XP_SIG('bells'));
```

The above code will output.

```
A wedding just happened.
```

Example #2 Detecting a wedding using network recieved signals.

```
<?php

xp_import('network');

$server = network\connect('0.0.0.0', ['port' => 8000]);
// Setup a server that emits a signal of recieved data
$server->on_read(function($signal){
    $read = trim($signal->socket->read());
    if ($read == null) {
      return false;
    }
    xp_emit(XP_SIG($read));
});

// Once a bride, groom and bell signals are emitted we emit the wedding.
$wedding = xp_complex_sig(function($signal){
    if (!isset($this->reset) || $this->reset) {
      $this->reset = false;
      $this->bride = false;
      $this->groom = false;
      $this->bells = false;
    }
    switch (true) {
      case $signal->compare(XP_SIG('groom')):
          $this->groom = true;
          break;
      case $signal->compare(XP_SIG('bride')):
          $this->bride = true;
          break;
      case $signal->compare(XP_SIG('bells')):
          $this->bells = true;
          break;
    }
    if ($this->groom && $this->bride && $this->bells) {
      $this->reset = true;
      return true;
    }
    return false;
});

xp_signal($wedding, function(){
    echo 'A wedding just happened.';
});

// Start the wait loop
xp_wait_loop();
```

The above code will output.

```
A wedding just happened.
```

Once the `bride`, `groom` and `bells` signals are emitted from the network connection the complex signal will emit the wedding.

Environment

XPSPL ships with the `xpspl` command for directly loading into an environment.

XPSPL understands the following.

```
usage: xpspl [-c|--config=<file>] [-d] [-h|--help] [-p|--passthru] [--test]
             [--test-cover] [-v|--version] [-j|--judy]
             <file>
Options:
  -c/--config   Load the giving file for configuration
  -d            XPSPL Debug Mode
  -h/--help     Show this help message.
  -j/--judy     Enable judy support
  -p/--passthru Ignore any subsequent arguments and pass to <file>.
  --test        Run the XPSPL unit tests.
  --test-cover  Run unit tests and generate code coverage.
  -v/--version  Displays current XPSPL version.
```

CHAPTER 2

Modules

XPSPL Bundled modules.

FTP Module

The FTP Module provides Non-Blocking FTP transfers for XPSPL.

Note: Currently only uploading files to a remote server is supported.

Installation

The FTP Module is bundled with XPSPL as of version 3.0.0.

Requirements

PHP

PHP FTP extension must be installed and enabled.

XPSPL

XPSPL **>= 3.0**

Configuration

The FTP Module has no runtime configuration options available.

Usage

Importing

```
<?php

xp_import('ftp');
```

Uploading Files

```
<?php

xp_import('ftp');

$files = ['/tmp/myfile_1.txt', '/tmp/myfile_2.txt'];
$server = [
    'hostname' => 'ftp.myhost.com',
    'username' => 'foo',
    'password' => 'bar'
];

$upload = ftp\upload($files, $server);

ftp\complete($upload, xp_null_exhaust(function(\ftp\SIG_Complete $sig){
    $file = $sig->get_file();
    echo sprintf('%s has uploaded'.PHP_EOL,
        $file->get_name()
    );
}));

ftp\failure($upload, xp_null_exhaust(function(\ftp\SIG_Failure $sig){
    $file = $sig->get_file();
    echo sprintf('%s has failed to upload'.PHP_EOL,
        $file->get_name()
    );
}));

ftp\finished($upload, function(\ftp\SIG_Finished $sig){
    echo 'FTP Transfer complete';
})
```

API

All functions and classes are under the `ftp` namespace.

ftp\upload(array $files, array $connection, [callable $callback = null])

Performs a non-blocking FTP upload of the given file(s).

When multiple files are given they will be uploaded simultaneously using separate connections to the given `$connection`.

The `$callback` will be called once the files begin uploading.

It is expected that the absolute path to the file will be given, failure to do so will cause unexpected behavior.

The connection array accepts,

•**hostname** - Hostname of the server to upload.

•**username** - Username to use when connecting.

•**password** - Password to use when connecting.

•**port** - Port number to connect on. *Default=21*

•**timeout** - Connection timeout in seconds. *Default=90*

Parameters

- **$files** (*array*) – Files to upload
- **$connection** (*array*) – FTP Connection options
- **$callback** (*callable*) – Function to call when upload begins

Return object SIG_Upload

Time Module

The Time module provides time based code execution for XPSPL.

Installation

The Time Module is bundled with XPSPL as of version 2.0.0.

XPSPL

XPSPL **>= 2.0**

Configuration

The Time Module has no runtime configuration options available.

Usage

Importing

```
<?php

xp_import('time');
```

Note: By default all times are in seconds.

Timeouts

Timeouts are functions executed once after a specific amount of time.

Timeouts are supported using the `awake` and xp_exhaust functions.

The below example demonstrates executing a timeout after 30 seconds.

```
<?php
// Import the time module.
xp_import('time');

// Define the exhaust rate of 1.
time\awake(30, xp_exhaust(1, function(){
    echo '30 seconds just passed';
});
```

Intervals

Intervals are functions executed continously after a specific amount of time.

The example below demonstrates an interval that runs every 30 seconds.

```
<?php

xp_import('time');

time\awake(30, function(){
    echo '30 seconds passed';
});
```

Milliseconds and Microseconds Instructions

The wait time period can be defined as seconds, milliseconds and microseconds.

Defining the instruction of time can be done by providing the `$instruction` to the `awake` function usign the time instruction constants.

- `TIME_SECONDS` Second instruction
- `TIME_MILLISECONDS` Millisecond instruction
- `TIME_MICROSECONDS` Microsecond instruction

The example below demonstrates executing timeouts in seconds, milliseconds and microseconds.

> **Warning:** Microsecond precision *CANNOT* be guaranteed.

```
<?php

xp_import('time');

// SECONDS
time\awake(30, function(){
    echo '30 seconds just passed';
}, TIME_SECONDS);
```

```
// MILLISECONDS
time\awake(1000, function(){
    echo '1000 milliseconds just passed';
}, TIME_MILLISECONDS);

// SECONDS
time\awake(1000, function(){
    echo '1000 microseconds just passed';
}, TIME_MICROSECONDS);
```

CRON Time

The CRON syntax is supported for executing signals based on the CRON syntax using the `CRON` api function.

The example below demonstrates executing a signal everyday at 12pm.

```
<?php
// CRON
time\CRON('12 * * * *', function(){
    echo 'It is 12 oclock!';
});
```

API

All functions and classes are under the `time` namespace.

time\awake($time, $callback, [$instruction = 4])
Wakes the system loop and runs the provided function.

Parameters

- **integer** – Time to wake in.
- **callable** – Callable function.
- **integer** – The time instruction. Default = Seconds

Return type array [signal, process]

time\CRON($cron, $process)
Wakes the system using the Unix CRON expressions.

If no priority is provided with the `` `$process` `` it is set to null.

Parameters

- **string** – CRON Expression
- **callable** – Callback function to run

Return type array [signal, process]

Network Module

The Network module provides non-blocking signal driven I/O.

Installation

The network Module is bundled with XPSPL.

Requirements

PHP

PHP Sockets extension must be installed and enabled.

XPSPL

XPSPL **>= 3.0**

Configuration

The Network Module has no runtime configuration options available.

API

network\connect($address, [$options = array()])

Creates a new socket connection.

Connection options.

- **port** - Default: null
- **domain** - Default: AF_INET
- **type** - Default: SOCK_STREAM
- **protocol** - Default: SOL_TCP

Parameters

- **string** – Address to make the connection on.
- **string** – Connection options
- **callback** – Function to call when connected

Return type object networkSocket

Usage

```
<?php

xp_import('network');
```

This demonstrates a basic network server that recieves connections on port 8000 sends "HelloWorld" to the connection and disconnects.

```php
<?php

xp_import('network');

$conn = network\connect('127.0.0.1', ['port' => 8000]);

$conn->on_connect(function($signal){
    $signal->socket->write('HelloWorld');
    $signal->socket->disconnect();
});
```

```php
<?php

xp_import('network');

$socket = network\connect('0.0.0.0', ['port' => '8000'], function(){
    echo "Server Running on " . $this->socket->get_address() . PHP_EOL;
});

$socket->on_connect(function(network\SIG_Connect $sig_connect) use ($socket){
    $sig_connect->socket->write("Welcome to the chat server".PHP_EOL);
    $sig_connect->socket->write("Enter your username : ");
});
$socket->on_read(null);
$socket->on_read(function(network\SIG_Read $sig_read) use ($socket){
    $clients = $socket->get_connections();
    $client = $clients[intval($sig_read->socket->get_resource())];
    // Strip any newlines from linux
    $sig_read->socket->_read_buffer();
    $content = implode("", explode("\r\n", $sig_read->socket->read()));
    // windows
    $content = implode("", explode("\n\r", $content));
    // On first connection read in the username
    if (!isset($client->username)) {
        $client->username = $content;
        $sig_read->socket->write("Welcome $content".PHP_EOL);
        foreach ($clients as $_client) {
            if ($_client != $sig_read->socket) {
                $_client->write(sprintf(
                    '%s has connected'.PHP_EOL,
                    $content
                ));
            }
        }
        $connected = [];
        foreach ($clients as $_client) {
            if (isset($_client->username)) {
                $connected[] = $_client->username;
            }
        }
        $sig_read->socket->write(sprintf(
            "%s User Online (%s)".PHP_EOL,
            count($connected),
            implode(", ", $connected)
        ));
    } else {
        foreach ($clients as $_client) {
            if ($_client != $sig_read->socket) {
```

```
                $_client->write(sprintf(
                    '%s : %s'.PHP_EOL,
                    $client->username,
                    $content
                ));
            }
        }
    }
});

$socket->on_disconnect(function() use ($socket){
    $clients = $socket->get_clients();
    $client = $clients[$sig_read->socket->get_resource()];
    foreach ($clients as $_client) {
        if ($_client != $sig_read->socket) {
            $_client->write(sprintf(
                '%s Disconnected'.PHP_EOL,
                $client->username
            ));
        }
    }
});
```

CHAPTER 3

Source

XPSPL is hosted on Github.

CHAPTER 4

Support

Have an issue?

File a bug report in Github issues.

CHAPTER 5

Author

XPSPL is designed and developed by Nickolas Whiting.

Index

X

xp_after() (built-in function), 3
xp_before() (built-in function), 4
xp_clean() (built-in function), 4
xp_complex_sig() (built-in function), 5, 33
xp_current_signal() (built-in function), 7
xp_delete_process() (built-in function), 8
xp_delete_signal() (built-in function), 8
xp_dir_include() (built-in function), 9
xp_emit() (built-in function), 10, 30
xp_erase_history() (built-in function), 12
xp_erase_signal_history() (built-in function), 13
xp_exhaust() (built-in function), 13
xp_get_signal() (built-in function), 24
xp_high_priority() (built-in function), 14
xp_import() (built-in function), 15
xp_listen() (built-in function), 15
xp_low_priority() (built-in function), 16
xp_null_exhaust() (built-in function), 17
xp_on_shutdown() (built-in function), 17
xp_on_start() (built-in function), 18
xp_priority() (built-in function), 18
xp_process() (built-in function), 19
xp_register_signal() (built-in function), 19
xp_set_signal_history() (built-in function), 20
xp_shutdown() (built-in function), 20
XP_SIG() (built-in function), 20
xp_signal() (built-in function), 20, 28
xp_signal_history() (built-in function), 22
xp_threaded_process() (built-in function), 23
xp_wait_loop() (built-in function), 24
XPSPL() (built-in function), 25
XPSPL_flush() (built-in function), 25