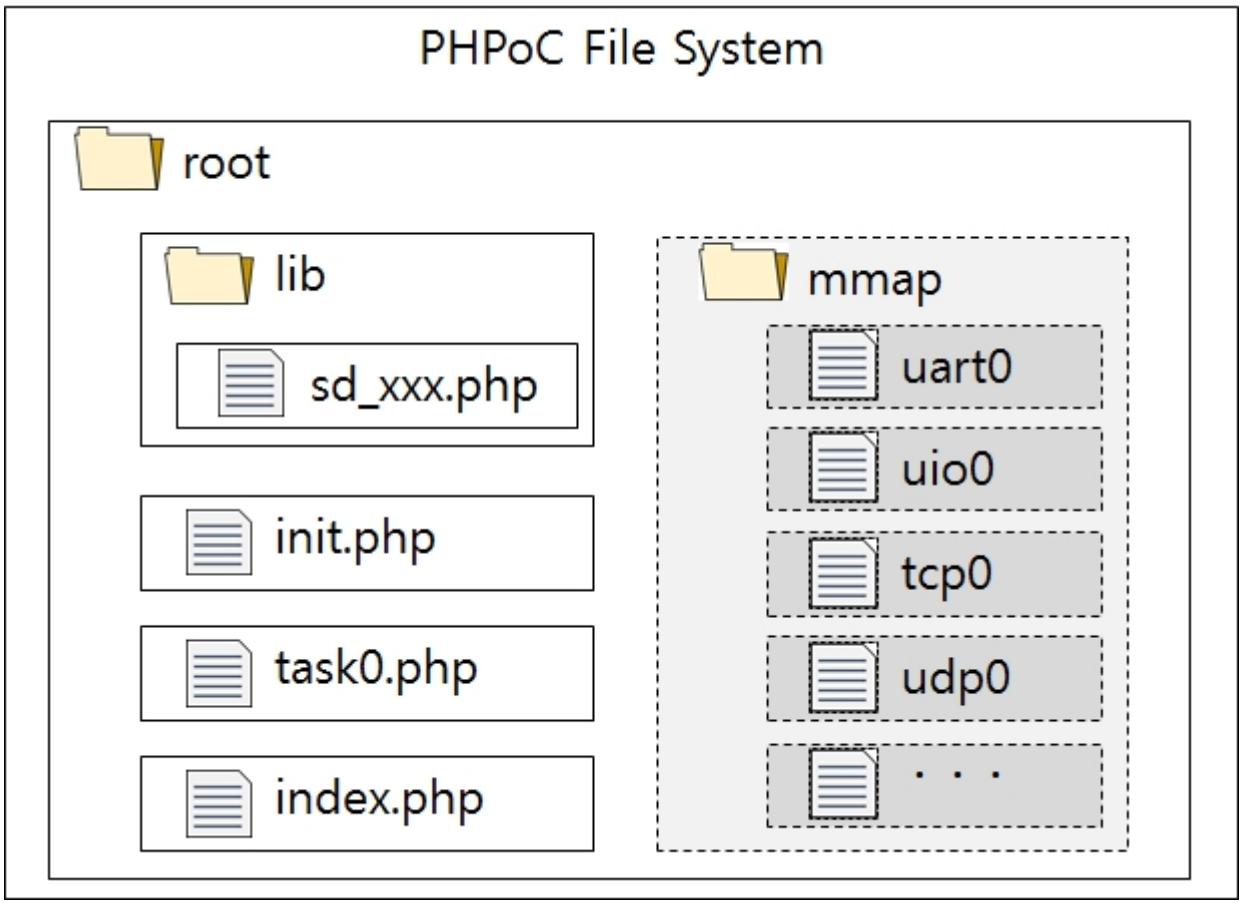
Device

Physical devices and software functions that PHPoC provides are called "device". Every device is provided as a special file form and can be used like a general file such as reading/writing files.

Path of Device Files

All device files of PHPoC are located in mmap (memory map) directory in root folder.



To access to a specific device, you should use a path like the example below.

```
/mmap/DEVICE_NAME
```

※ You can only access to the root, /lib and /mmap directory in this file system. In addition, users are not allowed to make or remove directories.

Types of Devices

PHPoC provides such types of devices below.

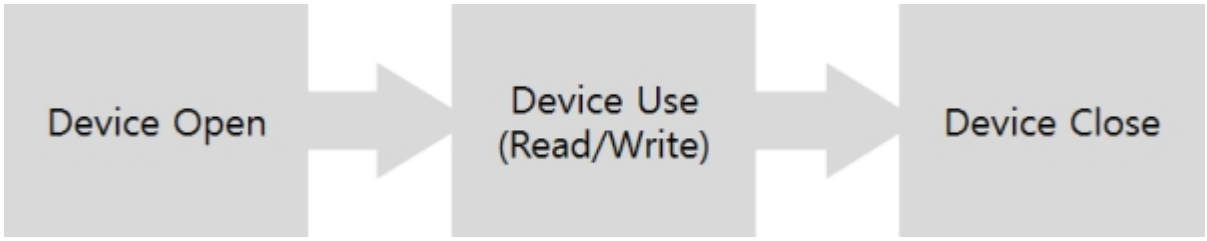
Division	Device Name
Hardware	Digital I/O(Input and Output), UART(Serial), NET(Network), ADC(Analog Input), I2C, SPI, HT(Hardware Timer), RTC(Real Time Clock)
Software	TCP, UDP, ST(Software Timer), System ENV, User ENV, User Memory, Non-volatile Memory

※ Types may be different according to products and version of firmware.

※ Refer to Appendix for more detailed device information about devices depending on the types of products.

Steps of Using Devices

General steps of using devices are as follows:



Opening Device

pid_open function is for opening devices. This function returns pid (Peripheral ID), which is an integer value, and this value is used for accessing to devices as a unique number.

Using Device

After opening successfully, device, which returned pid indicates, is ready to use. You can use it with functions such as pid_ioctl, pid_read and etc.

Closing Device

When device is not used anymore, it is needed to be closed by using pid_close function.

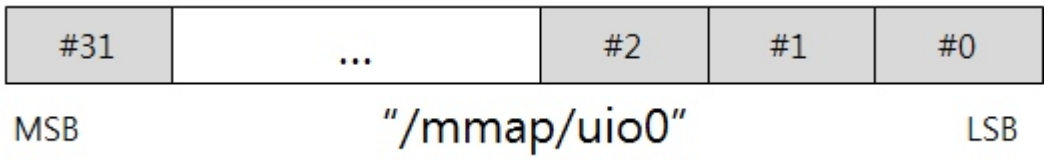
※ Caution: It is not possible to use a physical port by new device if the port has just been used, it is not possible to be used by new device until rebooted although the device was closed. In other words, a physical port cannot be used by more than two devices before it is initialized.

Overview

Digital I/O can be used to monitor digital inputs or control digital outputs. This device is also used to connect LED indicators showing system status.

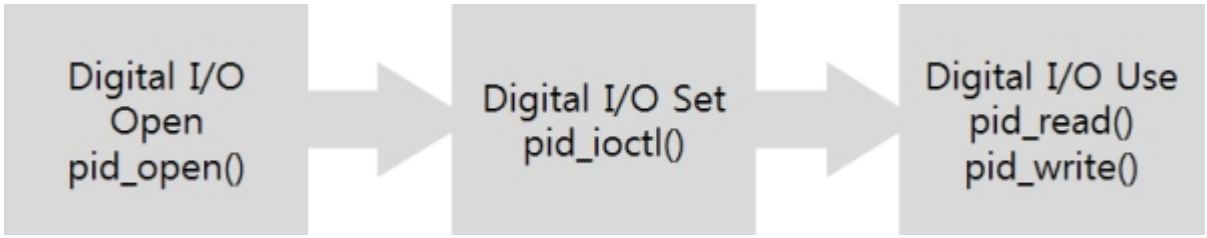
Digital I/O Structure

Every digital I/O port can have two different states which are High (or 1) and Low (or 0). Therefore, each port is matched to a binary digit as you can see below.



Steps of Using Digital I/O

General steps of using digital I/O ports are as follows:



Opening Digital I/O

To open digital I/O, pid_open function is required.

```
$pid = pid_open("/mmap/uio0");        // open UIO 0
```

※ Refer to Appendix for detailed digital I/O information depending on the types of products.

Setting Digital I/O

Before using digital I/O, setting each port is required. To set it, set command of pid_ioctl function is used.

```
pid_ioctl($pid, "set N1[-N2] mode TYPE");
```

N1 and N2 indicate a range of multiple ports. You can only use N1 in the case of setting a single port.

Setting to Input and Output

Available i/o types are as follows:

TYPE		Description
in		Digital Input
in_pu		Digital Input: Pull-Up
in_pd		Digital Input: Pull-Down
out	-	Digital Output
	low	Digital Output: default LOW
	high	Digital Output: default HIGH
out_pp	-	Digital Output: Push-Pull
	low	Digital Output: Push-Pull + default LOW
	high	Digital Output: Push-Pull + default HIGH
out_od	-	Digital Output: Open-Drain
	low	Digital Output: Open-Drain + default LOW
	high	Digital Output: Open-Drain + default HIGH

Input Port Pull-Up

Pull-Up makes a default state of a input port to HIGH. To do this, set the TYPE of the input port to in_pu.

Input Port Pull-Down

Pull-Down makes a default state of input port to LOW. To do this, set the TYPE of the input port to in_pd.

Output Port Push-Pull

Push-Pull is a basic output mode which makes a state of output port to HIGH when it is ON and LOW when it is OFF. To do this, set the TYPE of output port to out_pp.

Output Port Open-Drain

This can be used when you want to connect external power source to an output port. The state of output port will be LOW when it is OFF and UNKNOWN when it is ON if you do not connect any external power source while setting the port to open drain. Thus, you need to pull up this pin with an external resistor.

To do this, set the TYPE of output port to out_od.

Setting to LED

Digital I/O ports can be set to one of the LED types. Available types of LED are as follows:

TYPE	Description
led_sts	System Status LED
led_net0_act / led_net1_act	Activation of NET(net0 - wired, net1 - wireless) link LED: - successfully established network link: LOW - at the moment sending or receiving network data: HIGH
led_net0_link / led_net1_link	Network Link LED: connected to network - LOW
led_net0_rx / led_net1_rx	Network Receive LED: at the moment receiving data - LOW
led_net0_tx / led_net1_tx	Network Send LED: at the moment sending data - LOW

※ Each LED type cannot be set to two or more output pins.

example of setting digital I/O

```
$pid = pid_open("/mmap/uio0");                  // open UIO 0
pid_ioctl($pid, "set 0 mode in");              // set port 0 to input
pid_ioctl($pid, "set 1 mode in_pu");           // set port 1 to input: pull-up
pid_ioctl($pid, "set 2 mode in_pd");           // set port 2 to input: pull-down
pid_ioctl($pid, "set 3-6 mode out");           // set port 3 ~ 6 to output
// set port 7 ~ 9 to output with default high
pid_ioctl($pid, "set 7-9 mode out high");
// set port 10 to output with default low
pid_ioctl($pid, "set 10 mode out low");
// set port 11 to output: push-pull with default high
pid_ioctl($pid, "set 11 mode out_pp high");
// set port 12 to output: open-drain with default low
pid_ioctl($pid, "set 12 mode out_od low");
// set port 13 to network link LED
pid_ioctl($pid, "set 13 mode led_net0_link");
// set port 14 to network receive LED
pid_ioctl($pid, "set 14 mode led_net0_rx");
// set port 15 to network send LED
pid_ioctl($pid, "set 15 mode led_net0_tx");
```

Setting Output Lock

You can lock or unlock to control output ports by using pid_ioctl command. When output lock is enabled, output ports cannot be controlled before they are unlocked.

```
pid_ioctl($pid, "set N1[-N2] lock");     // lock
pid_ioctl($pid, "set N1[-N2] unlock");  // unlock
```

※ Caution: Digital I/O ports can be basically controlled. However, output lock is automatically enabled to ports which are shared with ST, UART, SPI and I2C if they are used.

Using Digital I/O

Reading states of Digital I/O

When reading status of digital I/O ports, you can get multiple states of them with pid_read function or a single state with pid_ioctl function. You can also read the type of a digital I/O port.

```
pid_read($pid, VALUE);            // read multiple states(in 32bits unit)
pid_ioctl($pid, "get N ITEM");    // read a single state(in a bit unit)
```

In the way of reading a single state, available ITEMs are as follows:

ITEM	Description	
mode	Return the port status in string type	I/O pin: "in", "out", "led_xxx" and so on
		pins while using by UART, SPI or I2C: "hdev"
		Designated to output pin of ST: "st_out"
input	Return the input port status in integer (0: LOW, 1: HIGH)	
output	Return the output port status in integer (0: LOW, 1: HIGH)	

example of reading multiple digital I/O states

The example below prints status of port 0 to 7 after setting them to input and getting the status.

```
$value = 0;
$pid = pid_open("/mmap/uio0");            // open UIO0
pid_ioctl($pid, "set 0-7 mode in_pu");  // set port 0 ~ 7 to input(pull-up)
pid_read($pid, $value);                  // read digital I/O status(32bits unit)
printf("0x%x₩r₩n", $value);              // output example: 0xffffffff
```

example of reading a single I/O state

The example below prints a state of port 0 of UIO0 after setting it to output and getting the mode and state.

```
$pid = pid_open("/mmap/uio0");              // open UIO0
pid_ioctl($pid, "set 0 mode out high");    // set port 0 to output
$mode = pid_ioctl($pid, "get 0 mode");     // read a digital I/O mode
$output = pid_ioctl($pid, "get 0 output"); // read a digital I/O state
printf("%s, %d₩r₩n", $mode, $output);      // output example: out, 1
```

※ When reading a port state with pid_ioctl function, you must use "get N input" if it is set to input port and use "get N output" if it is set to output port.

Writing Values to Digital I/O

When writing values to digital I/O ports, you can set a value to multiple ports with pid_write function or a single port with pid_ioctl function.

```
pid_write($pid, VALUE);                  // write to multiple ports(32 bits unit)
pid_ioctl($pid, "set N output TYPE");    // write to a single port(a bit unit)
```

example of writing values to multiple ports

The following example prints the states of digital I/O ports after setting 0 ~ 7 pins of UIO0 to output ports and writing a given value.

```
$value = 0;
$pid = pid_open("/mmap/uio0");                        // open UIO0
pid_ioctl($pid, "set 0-7 mode out");                  // set port 0 ~ 7 to output
pid_read($pid, $value);                               // read status
pid_write($pid, ($value & 0xffffff00) | 0x00000055); // write 0x00000055
pid_read($pid, $value);                               // read status
printf("0x%0x₩r₩n", $value);                          // output example: 0x00000055
```

example of writing a value to a single port

The following example prints a state of UIO0's port 0 after setting it to digital output with default LOW and writing HIGH.

```
$pid = pid_open("/mmap/uio0");              // open UIO0
pid_ioctl($pid, "set 0 mode out low");      // set port 0 to output(LOW)
pid_ioctl($pid, "set 0 output high");       // write HIGH
$output = pid_ioctl($pid, "get 0 output"); // read state of port 0
printf("%d₩r₩n", $output);                  // output: 1
```

example of setting output lock

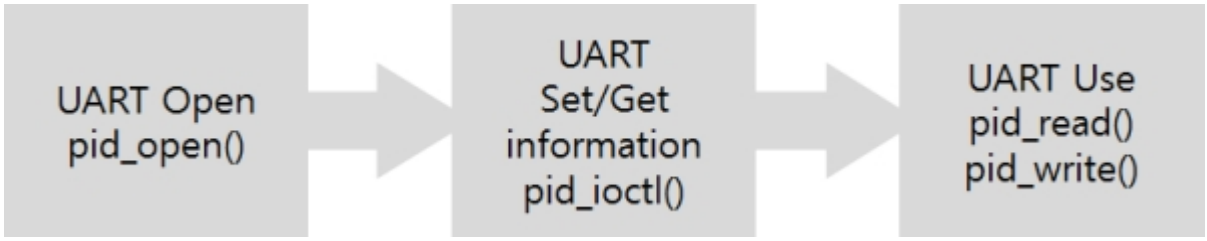
The following example shows the difference between locked and unlocked state of port 0.

```
$pid = pid_open("/mmap/uio0");               // open UIO0
pid_ioctl($pid, "set 0 mode out low");       // set port 0 to output(LOW)
pid_ioctl($pid, "set 0 lock");               // enable port 0 to output lock
pid_ioctl($pid, "set 0 output high");        // write HIGH to port 0
$output1 = pid_ioctl($pid, "get 0 output"); // read state of port 0
pid_ioctl($pid, "set 0 unlock");             // disable the output lock
pid_ioctl($pid, "set 0 output high");        // write HIGH to port 0 again
$output2 = pid_ioctl($pid, "get 0 output"); // read state of port 0
```

```
printf("%d, %d₩r₩n", $output1, $output2);    // output: 0, 1
```

Steps of Using UART

General steps of using UART ports are as follows:



Opening UART

To open UART, pid_open function is required.

```
$pid = pid_open("/mmap/uart0");      // opening UART 0
```

※ Refer to Appendix for detailed UART information depending on the types of products.

Setting UART

Before using UART, it needs to set parameters such as baud rate, data bit, stop bit, parity and flowcontrol by using "set" command of pid_ioctl function.

```
pid_ioctl($pid, "set ITEM VALUE");
```

ITEM means setting items and VALUE is possible value of the item.

Available UART Items

ITEM	VALUE	Description	Default Value
baud	ex) 9600	baud rate[bps]	19200
parity	0	no parity	0
	1	EVEN parity	
	2	ODD parity	
	3	MARK parity (always 1)	
	4	SPACE parity (always 0)	
data	8	8 data bit	8
	7	7 data bit(it can be only used with parity bit)	
stop	1	1 stop bit	1
	2	2 stop bit	
flowctrl	0	no flow control	0
	1	RTS/CTS hardware flow control	
	2	Xon/Xoff software flow control	
	3	TxDE flow control for RS485	

example of setting UART

```
$pid = pid_open("/mmap/uart0");     // open UART 0
pid_ioctl($pid, "set baud 9600");  // baud rate: 9600 bps
pid_ioctl($pid, "set parity 0");    // no parity
pid_ioctl($pid, "set data 8");      // data bit length: 8
pid_ioctl($pid, "set stop 1");      // stop bit length: 1
pid_ioctl($pid, "set flowctrl 0"); // no flow control
```

Getting Status of UART

To get various states of UART, get command of pid_ioctl function is required.

```
$return = pid_ioctl($pid, "get ITEM");
```

Available UART States

ITEM	Description	Return Value	Return Type
baud	baud rate[bps]	e.g. 9600	Integer
parity	parity	0 / 1 / 2 / 3 / 4	Integer
data	data bit[bit]	8 / 7	Integer
stop	stop bit[bit]	1 / 2	Integer
flowctrl	flowctrl	0 / 1 / 2 / 3	Integer
txbuf	size of send buffer[Byte]	e.g. 1024	Integer
txfree	free send buffer size[Byte]	e.g. 1024	Integer
rxbuf	size of receive buffer[Byte]	e.g. 1024	Integer
rxlen	received data size[Byte]	e.g. 10	Integer

example of getting UART states

Checking current information of UART is as follows:

```
$pid = pid_open("/mmap/uart0");                 // open UART 0
$baud = pid_ioctl($pid, "get baud");            // get baud rate
$parity = pid_ioctl($pid, "get parity");        // get parity
$data = pid_ioctl($pid, "get data");            // get data bit
$stop = pid_ioctl($pid, "get stop");            // get stop bit
$flowctrl = pid_ioctl($pid, "get flowctrl");    // get flow control mode
echo "baud = $baud₩r₩n";                        // output e.g.: baud = 9600
echo "parity = $parity₩r₩n";                    // output e.g.: parity = 0
echo "data = $data₩r₩n";                        // output e.g.: data = 8
echo "stop = $stop₩r₩n";                        // output e.g.: stop = 1
echo "flowctrl = $flowctrl₩r₩n";                // output e.g.: flowctrl = 0
```

Remaining Data Size in Send Buffer

Remaining data size in send buffer can be calculated as follows:

```
remaining data size in send buffer = size of buffer - free size of buffer
```

example

This example shows how to check remaining data size of send buffer.

```
$txlen = -1;
$data = "0123456789";
$pid = pid_open("/mmap/uart0");              // open UART 0
pid_ioctl($pid, "set baud 9600");            // baud rate: 9600 bps
pid_ioctl($pid, "set parity 0");             // parity: none
pid_ioctl($pid, "set data 8");               // data bit: 8
pid_ioctl($pid, "set stop 1");               // stop bit: 1
pid_ioctl($pid, "set flowctrl 0");           // flow control: none
pid_write($pid, $data);                      // write data to UART
while($txlen)
{
    $txbuf = pid_ioctl($pid, "get txbuf");   // get size of send buffer
    $txfree = pid_ioctl($pid, "get txfree"); // get remaining size of send buffer
    $txlen = $txbuf - $txfree;               // calculate remaining data size
    echo "tx len = $txlen₩r₩n";              // prints the size
    usleep(1000);
}
pid_close($pid);
```

Received Data Size

The following shows how to get received data size of UART.

```
$rxlen = pid_ioctl($pid, "get rxlen[ $string]");
```

Getting received data size with a string

If a string is specified after "get rxlen" command, pid_ioctl function returns 0 until the string comes into UART. If the specified string comes, it returns the whole data size including the string.

Remaining Size of Receive Buffer

Remaining size of receive buffer can be calculated as follows:

```
>remaining size of receive buffer = size of buffer - received data size
```

example

This example shows how to get remaining size of receive buffer.

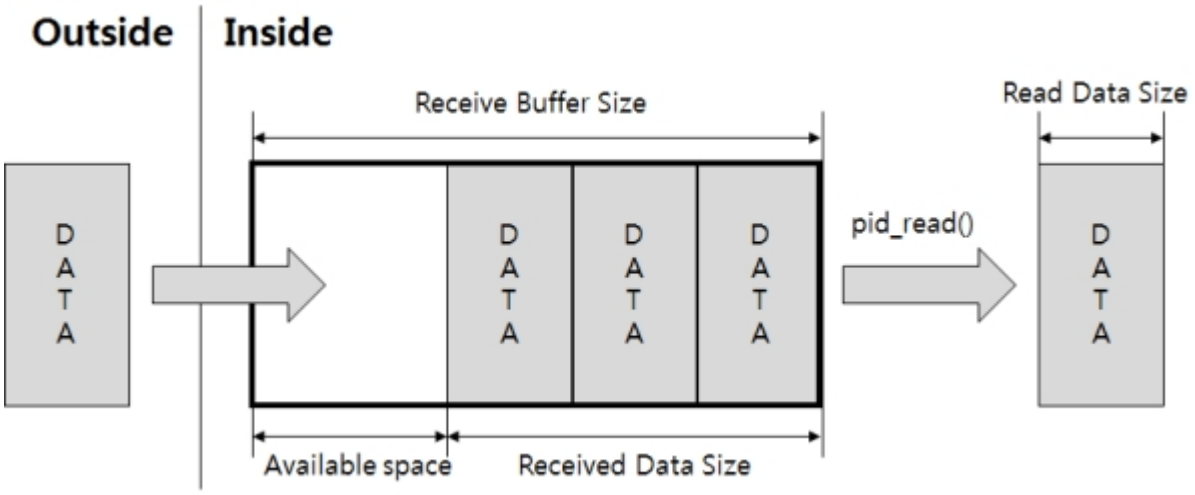
```
$rdata = "";
$pid = pid_open("/mmap/uart0");          // open UART 0
pid_ioctl($pid, "set baud 9600");        // baud rate: 9600 bps
pid_ioctl($pid, "set parity 0");         // parity: none
pid_ioctl($pid, "set data 8");           // data bit: 8
pid_ioctl($pid, "set stop 1");           // stop bit: 1
```

```
pid_ioctl($pid, "set flowctrl 0");       // flow control: none
$rxbuf = pid_ioctl($pid, "get rxbuf");  // get size of receive buffer
$rxlen = pid_ioctl($pid, "get rxlen");  // get received data size
$rxfree = $rxbuf - $rxlen;              // get remaining size of receive buffer
echo "rxfree = $rxfree₩r₩n";            // print the size
pid_close($pid);
```

Using UART

Reading Data

Data received from UART is stored in receive buffer. pid_read function is required to read the data.



The following shows how to use the pid_read function.

```
pid_read($pid, $var[, $len]);
```

Argument $var is a variable for saving the read data and $len is size of read data.

example

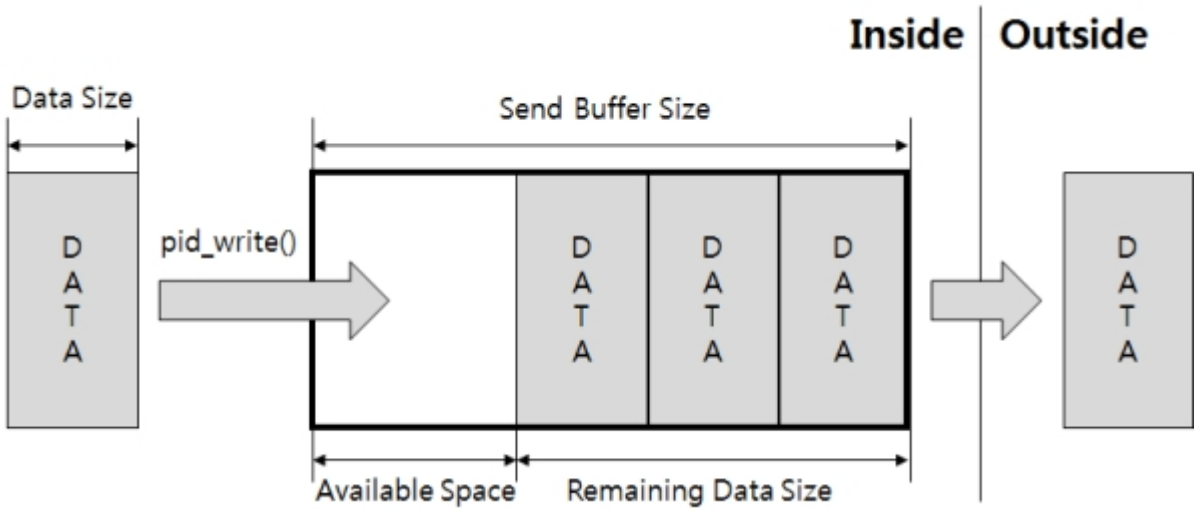
This example checks and prints received data to UART every second.

```
$rdata = "";
$pid = pid_open("/mmap/uart0");              // open UART 0
pid_ioctl($pid, "set baud 9600");           // baud rate: 9600bps
pid_ioctl($pid, "set parity 0");            // parity: none
pid_ioctl($pid, "set data 8");              // data bit: 8
pid_ioctl($pid, "set stop 1");              // stop bit: 1
$rxbuf = pid_ioctl($pid, "get rxbuf");      // get size of receive buffer
while(1)
{
    $rxlen = pid_ioctl($pid, "get rxlen");  // get size of received data
    $rx_free = $rxbuf - $rxlen;             // get remaining size
    echo "$rx_free / $rxbuf₩r₩n";           // print remaining size
    $len = pid_read($pid, $rdata, $rxlen);  // read data
    echo "len = $len / ";                   // print size of read data
    echo "rdata = $rdata₩r₩n";              // print read data
    sleep(1);
}
```

```
pid_close($pid);
```

Sending Data

Data, written by pid_write function, is stored in send buffer and transferred to the outside via UART.



The following shows how to use pid_write function.

```
pid_write($pid, $var[, $wlen]);
```

Argument $var is a variable containing data to send and $wlen is a size of sending data.

example

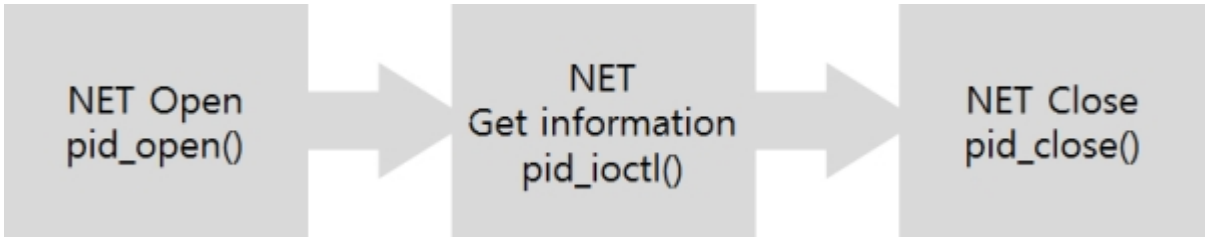
This example prints the remaining size of send buffer and length of sent data every second.

```
$sdata = "0123456789";
$pid = pid_open("/mmap/uart0");                 // open UART 0
pid_ioctl($pid, "set baud 9600");               // baud rate: 9600bps
$txbuf = pid_ioctl($pid, "get txbuf");          // get size of send buffer
while(1)
{
    $txfree = pid_ioctl($pid, "get txfree");  // get remaining size
    echo "txfree = $txfree₩r₩n";                // print remaining size
    $len = pid_write($pid, $sdata, $txfree);  // write data
    echo "len = $len₩r₩n";                      // print length of data sent
    sleep(1);
}
pid_close($pid);
```

The third argument of pid_write function means the length of writing data. The length of writing data should be less than the remaining data size of send buffer to avoid data loss. It is highly recommended to check remaining size of send buffer before sending data.

Steps of Using NET

General steps of using NET are as follows:



Opening NET

To open NET, pid_open function is required.

```
$pid = pid_open("/mmap/net0");          // opening NET 0
```

※ Refer to Appendix for detailed NET information depending on the types of products.

Getting Status of NET

To get a status of the NET port, get command of pid_ioctl function is required.

```
$return = pid_ioctl($pid, "get ITEM");
```

ITEM is a name of available states.

Available NET States

ITEM	Description	Return Value	Return Type
hwaddr	MAC Address	e.g. 00:30:f9:00:00:01	string
ipaddr	IP Address	e.g. 10.1.0.1	string
netmask	Subnet Mask	e.g.) 255.0.0.0	string
gwaddr	Gateway Address	e.g. 10.1.0.254	string
nsaddr	Name Server Address	e.g. 10.1.0.254	string
mode	10M Ethernet	10BASET	string
	100M Ethernet	100BASET	string
	WLAN Unavailable	""(an Empty String)	string
	WLAN Infrastructure	INFRA	string
	WLAN Ad-hoc	IBSS	string
	WLAN Soft AP	AP	string
speed	Ethernet Speed[Mbps]	0 / 10 / 100	integer
	WLAN Speed[100Kbps]	0 / 10 / 20 / 55 / 110 / 60 / 90 / 120 / 180 / 240 / 360 / 480 / 540	integer

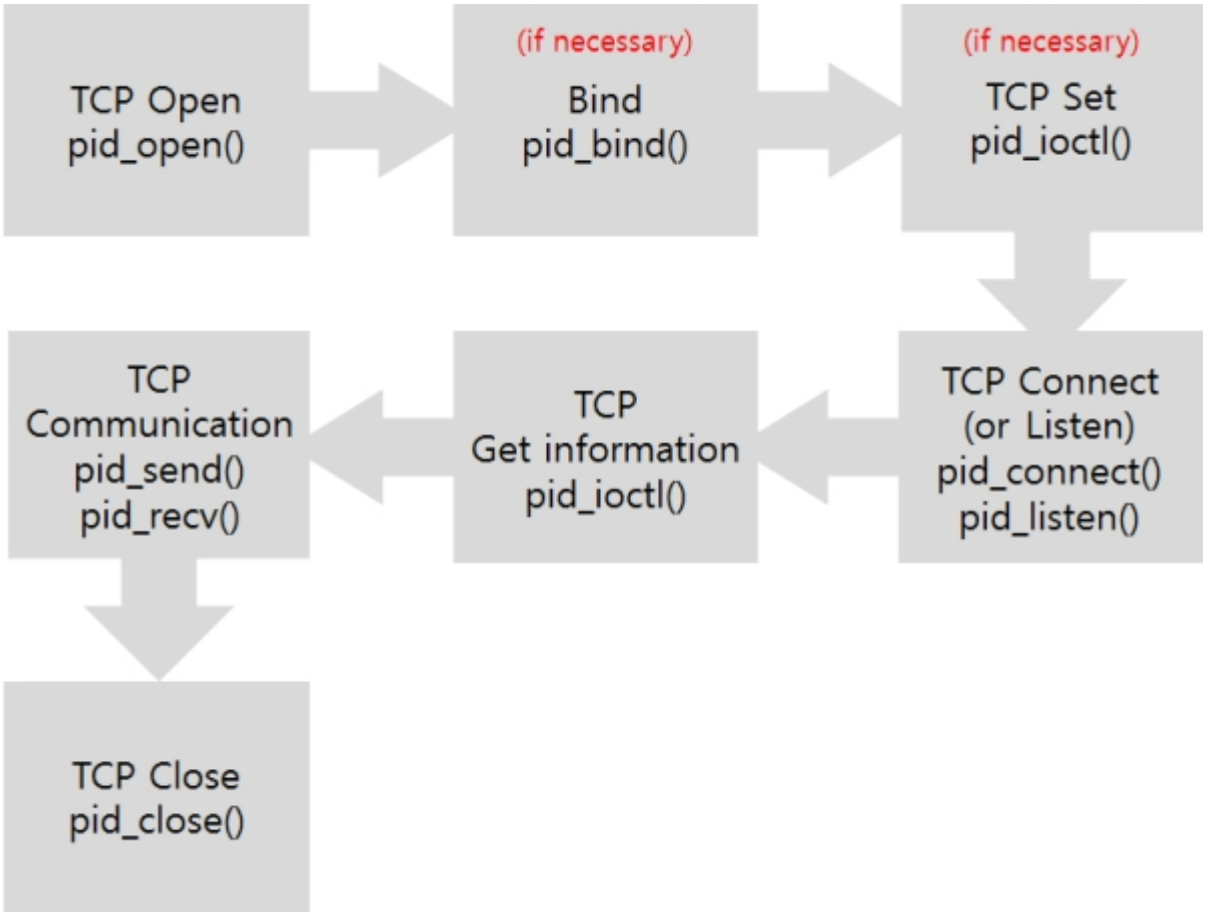
example of getting NET states

This example checks and prints various states of NET.

```
$pid = pid_open("/mmap/net1");                // open NET 1
echo pid_ioctl($pid, "get hwaddr"), "₩r₩n";  // get MAC address
echo pid_ioctl($pid, "get ipaddr"), "₩r₩n";  // get IP address
echo pid_ioctl($pid, "get netmask"), "₩r₩n"; // get subnet mask
echo pid_ioctl($pid, "get gwaddr"), "₩r₩n";  // get gateway address
echo pid_ioctl($pid, "get nsaddr"), "₩r₩n";  // get name server address
echo pid_ioctl($pid, "get mode"), "₩r₩n";    // get WLAN mode
echo pid_ioctl($pid, "get speed"), "₩r₩n";   // get WLAN speed
pid_close($pid);                              // close NET 1
```

Steps of Using TCP

General steps of using TCP are as follows:



※ In the case of setting a device to a TCP server, binding phase cannot be omitted.

Opening TCP

To open a TCP session, pid_open function is required.

```
$pid = pid_open("/mmap/tcp0");          // open TCP 0
```

※ Refer to Appendix for detailed TCP information depending on the types of products.

Setting TCP

Some parameters may be needed to set before using TCP. On SSL, SSH, TELNET or web socket communication, especially, SSL setting is required before connection by set command of pid_ioctl function.

```
pid_ioctl($pid, "set ITEM VALUE");
```

ITEM means setting items and VALUE is possible value of the item.

Available TCP Items

ITEM	VALUE		Description
nodelay	0		Enable Nagle algorithm
	1		Disable Nagle algorithm
api	ssl		Use SSL
	ssh		Use SSH server
	telnet		Use TELNET server
	ws		Use Web Socket server
ssl method	ssl3_client		SSL client (SSL 3.0)
	tls1_client		SSL client (TLS 1.0)
	ssl3_server		SSL server (SSL 3.0)
	tls1_server		SSL server (TLS 1.0)
ssh auth	accept		Accept SSH authorization
	reject		Reject SSH authorization
ws	path	PATH	Set path of web socket URI
	mode	0	Set data type of web socket: text
		1	Set data type of web socket: binary
	proto	PROTOCOL	Set protocol of web socket
	origin	ADDR	Specify a host to allow connection

TCP Nagle Algorithm is to improve effective data transmission by reducing the number of segments. Thus, it may accompany a little delay.

※ Note: Items by "set api" commands are only available on TCP 0 to 3. In addition, it is not possible to set another api mode once a TCP device is set to SSL or SSH before product reboots.

How to Use SSL

PHPoC can be an SSL server or client by "set api ssl" command. The following example shows how to use it as an SSL server.

example of SSL server

```
$port = 1470;                                   // port number
$pid = pid_open("/mmap/tcp0");                  // open TCP 0
pid_ioctl($pid, "set api ssl");                 // set api to SSL
pid_ioctl($pid, "set ssl method tls1_server");  // set SSL server mode
pid_bind($pid, "", $port);                      // binding
pid_listen($pid);                               // listen TCP connection
do
    $state = pid_ioctl($pid, "get state");
while(($state != SSL_CLOSED) && ($state != SSL_CONNECTED));

if($state == SSL_CONNECTED)
{
    echo "Connection has been established!₩r₩n";
    pid_close($pid);                            // close TCP connection
}
```

※ It is necessary to store a certification into PHPoC before you use it as a SSL server. Create or save a certificate to your product by PHPoC Debugger.

The following example shows how to use PHPoC as an SSL client.

example of SSL client

```
$addr = "10.1.0.2";                             // server's IP address
$port = 1470;                                   // server's port number
$pid = pid_open("/mmap/tcp0");                  // open TCP 0
pid_ioctl($pid, "set api ssl");                 // set api to SSL
pid_ioctl($pid, "set ssl method tls1_client");  // set SSL client mode
pid_connect($pid, $addr, $port);                // connect to TCP server
do
    $state = pid_ioctl($pid, "get state");
while(($state != SSL_CLOSED) && ($state != SSL_CONNECTED));

if($state == SSL_CONNECTED)
{
    echo "Connection has been established!₩r₩n";
    pid_close($pid);                            // close TCP connection
}
```

※ SSL communication may not be performed in case of lack of memory caused by increased memory usage of PHPoC.

How to Use TELNET

PHPoC can be set as a TELNET server by using "set api telnet" command. The following is an example of a TELNET server.

example of a TELNET server

```
$port = 23;                             // port number
$pid = pid_open("/mmap/tcp0");          // open TCP 0
pid_ioctl($pid, "set api telnet");      // set api to TELNET
pid_bind($pid, "", $port);              // binding
pid_listen($pid);                       // listen TCP connection
do
    $state = pid_ioctl($pid, "get state");
while(($state != TCP_CLOSED) && ($state != TCP_CONNECTED));

if($state == TCP_CONNECTED)
{
    pid_send($pid, "Welcome to PHPoC TELNET server₩r₩n");
    echo "Connection has been established!₩r₩n";
    pid_close($pid);                    // close TCP connection
}
```

In the example above, PHPoC listens TELNET connection from clients. After connection is established, it prints a welcome message and close the connection.

※ If you want to test this example, open a TELNET client program on PC such as Tera Term and try connecting to PHPoC TELNET server.

※ If you want to do authentication including user identification, you should implement it in user script.

How to Use SSH Server

PHPoC can be set to an SSH server by using "set api ssh" command. The following example shows how to make an SSH server.

example of SSH server

```
$port = 22;                         // port number
$pid = pid_open("/mmap/tcp0");      // open TCP 0
pid_ioctl($pid, "set api ssh");     // set api to SSH
pid_bind($pid, "", $port);          // binding
pid_listen($pid);                   // listen TCP connection
while(1)
{
    $state = pid_ioctl($pid, "get state");
    if($state == SSH_AUTH)
    {
        $username = pid_ioctl($pid, "get ssh username");
        $password = pid_ioctl($pid, "get ssh password");
        echo "$username / $password₩r₩n";
        pid_ioctl($pid, "set ssh auth accept");
    }
    if($state == SSH_CONNECTED)
    {
        pid_send($pid, "Welcome to PHPoC SSH server₩r₩n");
        echo "Connection has been established!₩r₩n";
        pid_close($pid);
        break;
    }
}
```

In the example above, PHPoC listens SSH connection from clients. After connection is established, it prints a username and a password from client. After that, it prints a welcome message and close the connection.

※ If you want to test this example, open a SSH client program on PC such as Tera Term and try connecting to PHPoC SSH server.

※ Authentication process including user identification should be implemented in user script.

How to Use Web Socket Server

PHPoC can be a web socket server by using "set api ws" command. The following example shows how to use web socket server.

example of web socket server

This example listens TCP connection from clients. After connection is established, PHPoC prints data which is received from clients including the count of receiving data.

```
$pid = pid_open("/mmap/tcp0");                  // open TCP 0
pid_ioctl($pid, "set api ws");                  // set api to web socket
pid_ioctl($pid, "set ws path WebConsole");     // set URI path: /WebConsole
pid_ioctl($pid, "set ws mode 0");               // set transmission mode: text
//pid_ioctl($pid, "set ws origin 10.1.0.1");  // specify a host to allow connection
pid_ioctl($pid, "set ws proto text.phpoc");    // protocol: text.phpoc
pid_bind($pid, "", 0);                          // binding: default(80)

while(1)
{
  if(pid_ioctl($pid, "get state") == TCP_CLOSED)
    pid_listen($pid);                           // listen TCP connection

  pid_send($pid, "hello, world!₩r₩n");         // send data
  sleep(1);
}
pid_close($pid);
```

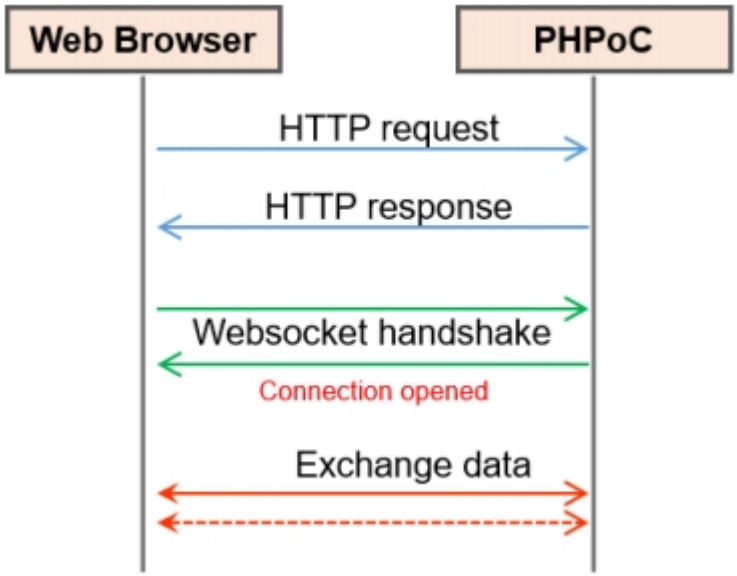
The following is a simple source code of web page which can be used for as a web socket client to connect with PHPoC web socket server above.

```
<html>
<head>
<title>PHPoC / <?echo system("uname -i")?></title>
<meta name="viewport" content="width=device-width, initial-scale=0.7">
<style>
body { text-align:center; }
textarea { width:400px; height:400px; padding:10px; font-family:courier; font-size:14px; }
 </style>
<script>
var ws;
var wc_max_len = 32768;
function ws_onopen()
{
    document.getElementById("ws_state").innerHTML = "OPEN";
    document.getElementById("wc_conn").innerHTML = "Disconnect";
}
function ws_onclose()
{
```

```
    document.getElementById("ws_state").innerHTML = "CLOSED";
    document.getElementById("wc_conn").innerHTML = "Connect";

    ws.onopen = null;
    ws.onclose = null;
    ws.onmessage = null;
    ws = null;
}
function wc_onclick()
{
    if(ws == null)
    {
        ws = new WebSocket("ws://<?echo _SERVER("HTTP_HOST")?>/WebConsole", "text.phpoc");
        document.getElementById("ws_state").innerHTML = "CONNECTING";

        ws.onopen = ws_onopen;
        ws.onclose = ws_onclose;
        ws.onmessage = ws_onmessage;
    }
    else
        ws.close();
}
function ws_onmessage(e_msg)
{
    e_msg = e_msg || window.event; // MessageEvent

    var wc_text = document.getElementById("wc_text");
    var len = wc_text.value.length;

    if(len > (wc_max_len + wc_max_len / 10))
        wc_text.innerHTML = wc_text.value.substring(wc_max_len / 10);

    wc_text.scrollTop = wc_text.scrollHeight;
    wc_text.innerHTML += e_msg.data;
}
function wc_clear()
{
    document.getElementById("wc_text").innerHTML = "";
}
</script>
</head>
<body>

<h2>
<p>
Web Console : <span id="ws_state">CLOSED</span><br>
</p>
<textarea id="wc_text" readonly="readonly"></textarea><br>
<button id="wc_conn" type="button" onclick="wc_onclick();">Connect</button>
<button id="wc_clear" type="button" onclick="wc_clear();">Clear</button>
</h2>

</body>
</html>
```

in the above example, both web socket server (php script) and client (javascript) is implement on

PHPoC but web socket server is executed on PHPoC and web socket client is executed on web browser. The below image show the working flow of the above example.



※ You can make more powerful web interface by modifying the above web socket client script (in index.php) and web server script (in task0.php).

※ It is required to use a browser which supports web socket.

TCP Connection

TCP Client (Active Connection)

Active connection means sending a TCP connection request packet to a TCP server and this host is called TCP client. To perform TCP client, pid_connect function is used.

```
pid_connect($pid, $addr, $port);
```

Argument $addr is an IP address of a TCP server and $port is a port number.

example of TCP client

```
$pid = pid_open("/mmap/tcp0");   // open TCP
$addr = "10.1.0.2";              // IP address of TCP server
$port = 1470;                    // TCP port
pid_connect($pid, $addr, $port); // active TCP connection
sleep(25);
pid_close($pid);
```

TCP Server (Passive Connection)

Passive connection means listening a TCP connection request packet from a TCP client and this host is called TCP server. To perform TCP server, pid_bind and pid_listen function are required.

```
pid_bind($pid, "", $port);
pid_listen($pid[, $backlog]);
```

Argument $port is a TCP port number.

example of TCP Server

```
$pid = pid_open("/mmap/tcp0");  // open TCP
$port = 1470;                   // TCP port number
pid_bind($pid, "", $port);      // bind with the port number
pid_listen($pid);               // passive TCP connection
sleep(25);
pid_close($pid);
```

Getting Status of TCP

To get states of TCP, get command of pid_ioctl function is required.

```
$return = pid_ioctl($pid, "get ITEM");
```

Available TCP States

ITEM	Description	Return Value	Return Type
state	TCP session is closed	TCP_CLOSED	integer
	TCP session is connected	TCP_CONNECTED	integer
	TCP session waits for connection	TCP_LISTEN	integer
	SSL session is closed	SSL_CLOSED	integer
	SSL session is connected	SSL_CONNECTED	integer
	SSL session waits for connection	SSL_LISTEN	integer
	SSL session is closed	SSH_CLOSED	integer
	SSL session is connected	SSH_CONNECTED	integer
	SSL session waits for connection	SSH_LISTEN	integer
	SSL authentication is completed	SSH_AUTH	integer
srcaddr	local IP address	e.g. 192.168.0.1	string
srcport	local port number	e.g. 1470	integer
dstaddr	peer IP address	e.g. 192.168.0.2	string
dstport	peer TCP number	e.g. 1470	integer
txbuf	size of send buffer[Byte]	e.g. 1152	integer
txfree	remaining send buffer size[Byte]	e.g. 1152	integer
rxbuf	size of receive buffer[Byte]	e.g. 1068	integer
rxlen	received data size[Byte]/td>	e.g. 200	integer
ssh username	SSH user name	e.g. user	string
ssh password	SSH password	e.g. password	string

TCP Session Status

Checking status of connection on TCP is very important because TCP data communication is made after the connection phase. There are three session states: TCP_CLOSED when session is not connected, TCP_CONNECTED when session is connected and TCP_LISTEN when TCP server is listening connection. SSL and SSH have also these three states: SSL_CLOSED, SSL_CONNECTED and SSL_LISTEN and SSH has an additional state about authentication (SSH_AUTH). The following shows how to get states of session.

```
$state = pid_ioctl($pid, "get state");
```

※ An unknown value, which is not listed in the table above, could be returned if you try to get a state when PHPoC is connecting or closing connection. Note that it is not recommended to use these values in your script because it might be changed in the future.

Remaining Data Size in Send Buffer

Remaining data size in send buffer can be calculated as follows:

> remaining data size in send buffer = size of buffer - remaining size of buffer

example

In this example, PHPoC sends 8 bytes data to a server right after TCP connection is established. While sending the data, PHPoC prints remaining data size in send buffer.

```
$tx_len = -1;
$pid = pid_open("/mmap/tcp0");                  // open TCP 0
do
{
    pid_connect($pid, "10.1.0.2", 1470);      // TCP active connection
    usleep(500000);
}
while(pid_ioctl($pid, "get state") != TCP_CONNECTED);
pid_send($pid, "01234567");                     // send 8 bytes
while($tx_len && (pid_ioctl($pid, "get state") == TCP_CONNECTED))
{
    $txbuf = pid_ioctl($pid, "get txbuf");    // get the size of send buffer
    // get the empty size of send buffer
    $txfree = pid_ioctl($pid, "get txfree");
    // calculate the size of remaining data in send buffer
    $tx_len = $txbuf - $txfree;
    echo "tx len = $tx_len₩r₩n";               // print the result
    usleep(10000);
}
pid_close($pid);                              // close TCP
```

Received Data Size

The following shows how to get the received data size from TCP socket.

```
$rxlen = pid_ioctl($pid, "get rxlen[ $string]");
```

Getting the received data size with string

If a string is specified after "get rxlen" command, pid_ioctl function returns 0 until the string comes into TCP socket. If the specified string comes, the function returns the whole data size including the string.

Remaining Size of Receive Buffer

Remaining size of receive buffer can be calculated as follows:

> remaining size of receive buffer = size of buffer - size of received data

example

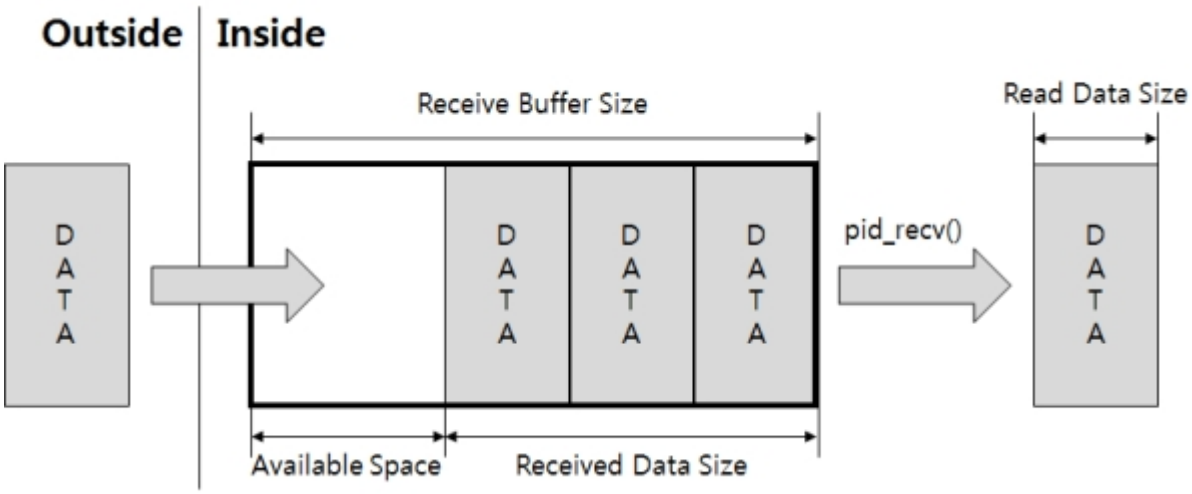
This example shows how to get the remaining size of receive buffer.

```
$rx_free = 1068;
$pid = pid_open("/mmap/tcp0");                  // open TCP 0
do
{
    pid_connect($pid, "10.1.0.2", 1470);      // TCP active connection
    usleep(500000);
}
while(pid_ioctl($pid, "get state") != TCP_CONNECTED);

while(($rx_free > 500) && (pid_ioctl($pid, "get state") == TCP_CONNECTED))
{
    $rxbuf = pid_ioctl($pid, "get rxbuf");    // get the size of receive buffer
    $rxlen = pid_ioctl($pid, "get rxlen");    // get the size of received data
    // calculate the available space of receive buffer
    $rx_free = $rxbuf - $rxlen;
    echo "rx free = $rx_free₩r₩n";             // print the result
    sleep(1);
}
pid_close($pid);                              // close TCP
```

TCP Communication

Receiving TCP Data

Data received from network via TCP is stored in receive buffer. pid_recv function is required to read the data.



The following shows how to use pid_recv function.

```
pid_recv($pid, $value[, $len]);
```

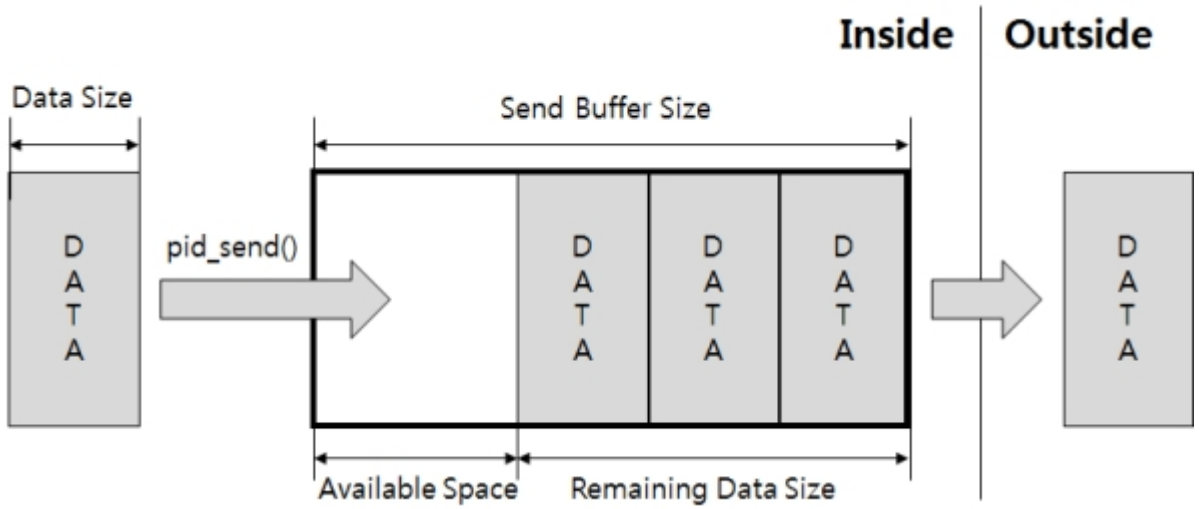
example

This example checks and prints the received TCP data every second.

```
$rdata = "";
$pid = pid_open("/mmap/tcp0");              // open TCP 0
pid_connect($pid, "10.1.0.2", 1470);        // TCP active connection
do
{
    sleep(1);
    $state = pid_ioctl($pid, "get state");  // get TCP session state
    $rxlen = pid_ioctl($pid, "get rxlen");  // get received data size
    $rlen = pid_recv($pid, $rdata, $rxlen); // receive data
    echo "rlen = $rlen / ";                 // print received data size
    echo "rdata = $rdata₩r₩n";              // print received data
    if($rlen)
        $rdata = "";                        // flush receive buffer
}
while($state == TCP_CONNECTED);
pid_close($pid);
```

Sending TCP Data

Data sent by pid_send function is stored in send buffer and transferred to the network via TCP.



The following shows how to use pid_send function.

```
pid_send($pid, $value[, $len]);
```

example

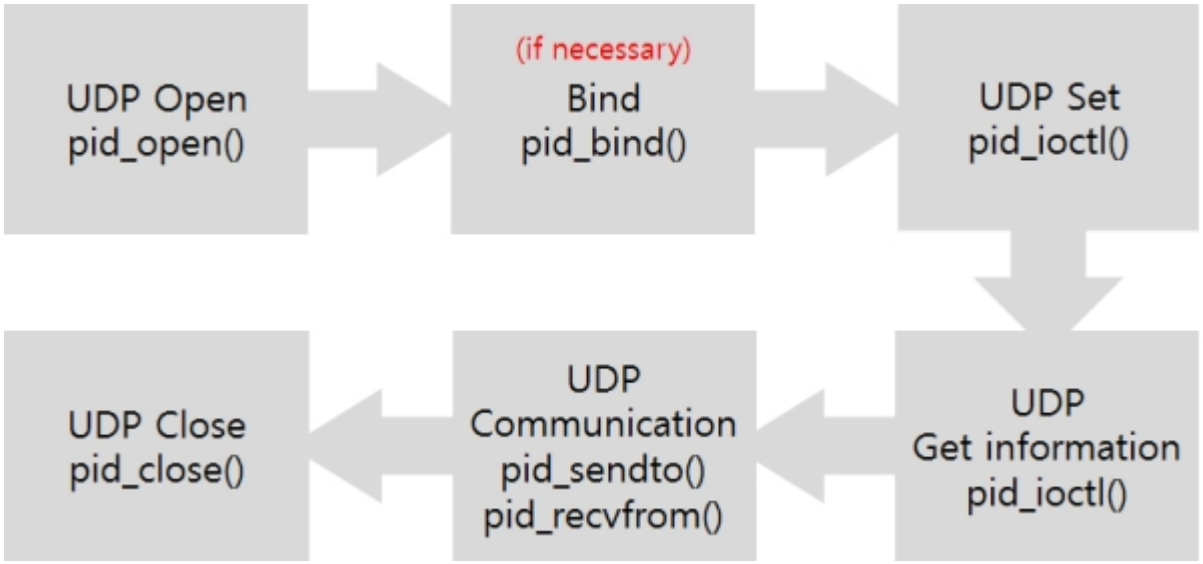
This example sends data to network via TCP, checking the available space of send buffer every second.

```
$sdata = "0123456789";
$pid = pid_open("/mmap/tcp0");                  // open TCP 0
pid_connect($pid, "10.1.0.2", 1470);            // TCP active connection
do
{
    sleep(1);
    $state = pid_ioctl($pid, "get state");      // get session state
    // get available space of send buffer
    $txfree = pid_ioctl($pid, "get txfree");
    $tx_len = pid_send($pid, $sdata, $txfree);  // send data
    echo "tx len = $tx_len₩r₩n";                // print size of send data
}
while($state == TCP_CONNECTED);
pid_close($pid);
```

The third argument of pid_send function means the length of sending data. The length of sending data should be less than the remaining data size of send buffer to avoid data loss. It is highly recommended to check the remaining size of send buffer before sending data.

Steps of Using UDP

General steps of using UDP are as follows:



※ Binding socket can be omitted if there is no requirement of prior setting or data transmission.

Opening UDP

To open UDP, pid_open function is required.

```
$pid = pid_open("/mmap/udp0");        // open UDP 0
```

Refer to Appendix for detailed UDP information depending on the types of products.

Binding

Binding which uses pid_bind function is required to receive data from network via UDP.

```
$pid = pid_bind($pid, $addr, $port);
```

Argument $addr is an IP address and $port is port number to bind. When empty string("") is specified to the IP address, PHPoC assumes the value is the current local IP address.

※ Empty string ("") value is the only option for $addr argument of function bind.

example of binding

```
$pid = pid_open("/mmap/udp0");    // open UDP
$port = 1470;                     // UDP port number
pid_bind($pid, "", $port);        // binding
```

Setting UDP

A destination IP address and port number can be specified before sending UDP data. If so, these parameters can be omitted in fourth and fifth argument of pid_sendto function. Set command of pid_ioctl function is required to set UDP.

```
pid_ioctl($pid, "set ITEM VALUE");
```

ITEM means setting items and VALUE is possible value of the item.

Available UDP Items

ITEM	VALUE	Description
dstaddr	e.g. 10.1.0.2	destination IP address
dstport	e.g. 1470	destination port number

example of setting UDP

```
$pid = pid_open("/mmap/udp0");              // open UDP 0
pid_bind($pid, "", 1470);                   // binding
pid_ioctl($pid, "set dstaddr 10.1.0.2");  // destination IP address
pid_ioctl($pid, "set dstport 1470");       // destination port number
```

Getting UDP Status

To get status of UDP, get command of pid_ioctl function is required.

```
$return = pid_ioctl($pid, "get ITEM");
```

Available UDP States

ITEM	Description	Return Value	Return Type
srcaddr	source IP address	e.g. 192.168.0.1	string
srcport	source port number	e.g. 1470	integer
dstaddr	destination IP address	e.g. 192.168.0.2	string
dstport	destination port number	e.g. 1470	integer
rxlen	received data size[Byte]	e.g. 200	integer

Received Data Size

To get received data size, "get rxlen" command of pid_ioctl function is required.

```
$rxlen = pid_ioctl($pid, "get rxlen");
```

example

This example is closed after printing received data size if data comes from network while checking periodically whether there is data or not.

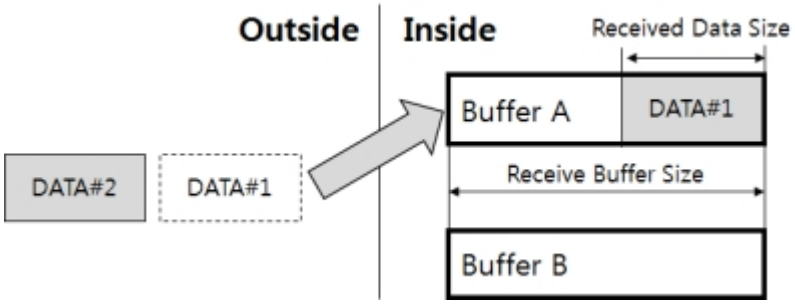
```
$rbuf = "";
$pid = pid_open("/mmap/udp0");              // open UDP 0
pid_bind($pid, "", 1470);                   // binding
do
{
    $rxlen = pid_ioctl($pid, "get rxlen");  // get received data size
    if($rxlen)
    {
        pid_recvfrom($pid, $rbuf, $rxlen);  // receive data
        echo "$rxlen bytes₩r₩n";            // print size of received data
    }
    usleep(100000);
}while($rxlen == 0);                        // while receiving no data
pid_close($pid);
```

UDP Communication

Receiving UDP Data

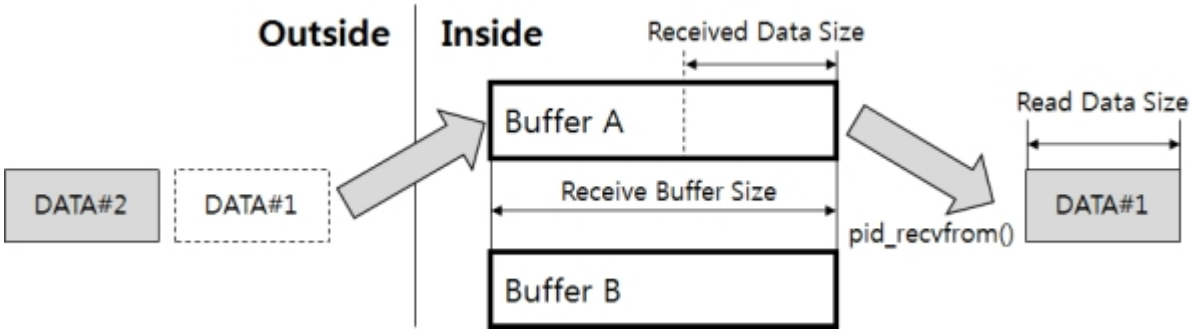
To receive data from network via UDP, pid_recvfrom function is required. There are two receive buffers of UDP and the following shows how they works. □
※ Refer to Appendix for information about UDP receive buffer size depending on the types of products.

receiving UDP data from network



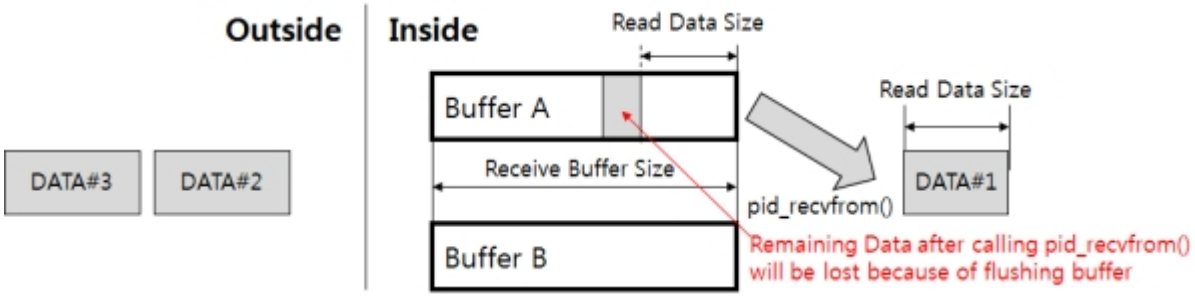
reading UDP data from receive buffer

After reading data from receive buffer by calling pid_recvfrom function, PHPoC flushes the buffer.



reading data size less than received data size

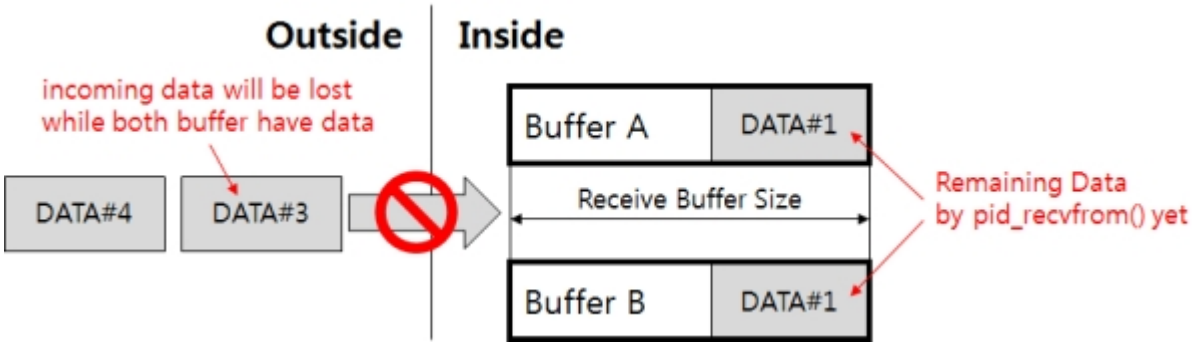
Remaining data after reading will be lost by flushing receive buffer.



losing data by no available receive buffer

While each of two receive buffers has data which have unread data, subsequent data from network

cannot be received. Therefore, it is recommended to read data as soon as possible in received buffer right after checking received data size.



example

This example prints received UDP data, checking if there is data comes from network every second.

```
$rbuf = "";
$pid = pid_open("/mmap/udp0");              // open UDP 0
pid_bind($pid, "", 1470);                   // binding
while(1)                                    // infinite loop
{
    $rxlen = pid_ioctl($pid, "get rxlen");  // get received data size
    if($rxlen)
    {
        pid_recvfrom($pid, $rbuf, $rxlen);  // receive data
        echo "$rbuf₩r₩n";                   // print received data
    }
    usleep(100000);
}
```

Sending UDP Data

To send UDP data, pid_sendto function is required.

example of sending UDP data

```
$sdata = "01234567";
$pid = pid_open("/mmap/udp0");                              // open UDP 0
$slen = pid_sendto($pid, $sdata, 8, 0, "10.1.0.2", 1470);  // send data
echo "slen = $slen₩r₩n";                                    // print size of send data
pid_close($pid);
```

Steps of Using ST

General steps of using ST are as follows:

Opening ST

To open ST, pid_open function is required.

```
$pid = pid_open("/mmap/st0");   // open ST 0
```

※ Refer to Appendix for detailed ST information depending on the types of products.

Setting and Using ST

To use ST, pid_ioctl function is required. There are four modes.

Mode	Description
Free mode	normal counter mode
Output Pulse mode	mode to output pulse signal through a specified pin
Output Toggle mode	mode to output toggle signal through a specified pin
Output PWM mode	mode to output infinite pulse through a specified pin

Common Commands

Commands listed in the table below are used in all modes of ST.

Command	Sub Command			Description
set	mode	free		set mode: free
		output	pulse	set mode: output pulse
			toggle	set mode: output toggle
			pwm	set mode: output infinite pulse
	div	sec		set unit: second
		ms		set unit: millisecond
		us		set unit: microsecond
reset	-			reset
get	state			get current state
start	-			start
stop	-			stop

Set Mode

ST provides both normal counter mode (free mode) and output signal mode. There are three output modes and those are pulse, toggle and pwm. The pwm is infinite pulse mode. Default value of ST mode is free mode. The following table shows how to set ST to each mode.

Mode	Syntax
free	pid_ioctl($pid, "set mode free");
pulse	pid_ioctl($pid, "set mode output pulse");
toggle	pid_ioctl($pid, "set mode output toggle");
pwm	pid_ioctl($pid, "set mode output pwm");

Set Unit

ST provides three units as follows. The default value is millisecond.

Unit	Syntax
second	pid_ioctl($pid, "set div sec");
millisecond	pid_ioctl($pid, "set div ms");
microsecond	pid_ioctl($pid, "set div us"");

Reset

This command immediately stops operation of ST and reset.

Command	Syntax
reset	pid_ioctl($pid, "reset");

Get State

This command gets the current state of ST.

Command	Syntax
get state	pid_ioctl($pid, "get state");

Return values of this command are as follows:

Return Value	Description
0	Stop
1 ~ 5	Running

Start

This command starts ST.

Command	Syntax
start	pid_ioctl($pid, "start");

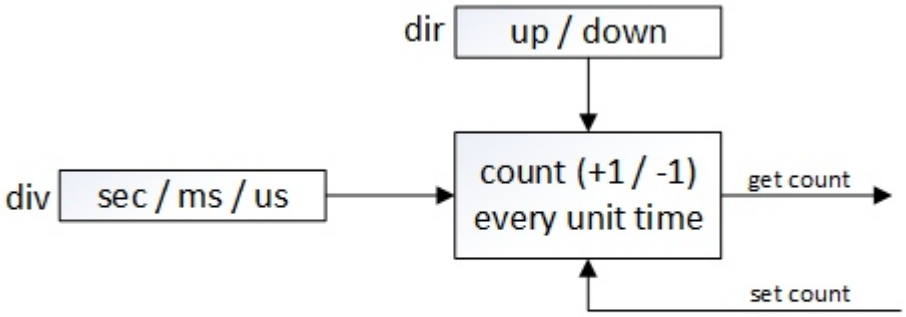
Stop

This command immediately stops operation of ST. In output modes, state of output pin keeps the current state.

Command	Syntax
stop	pid_ioctl($pid, "stop");

Free Mode

Free mode is normal counter mode of ST.



ST(free) Block Diagram

Available commands of pid_ioctl function in free mode are as follows:

Command	Sub Command		Description
set	mode	free	set mode: free mode
	div	sec	set unit: second
		ms	set unit: millisecond
		us	set unit: microsecond
	dir	up	set counter direction: up counter
		down	set counter direction: down counter
	count	[T]	set the starting count value in down counter mode
reset	-		reset
get	count		get count value
	state		get current state
start	-		start
stop	-		stop

Set Counter Direction

ST can be used as both up counter and down counter. Default value of this item is up counter.

Direction	Syntax
up counter	pid_ioctl($pid, "set dir up");
down counter	pid_ioctl($pid, "set dir down");

Set Count

When ST is down counter mode, the starting value of counter can be set.
How to set count value is as follows:

Command	Syntax
set count	pid_ioctl($pid, "set count T");

Note that up counter always starts from 0 and is not affected by "set count" command. Valid ranges for T in down counter are as follows:

Unit	Valid Range for T
Microsecond	0 ~ (2^{63} - 1)
milisecond	0 ~ (2^{63} - 1) / 1,000
second	0 ~ (2^{63} - 1) / 1,000,000

Get Count Value

Command "get count" returns a current count value.

Command	Syntax
get count	pid_ioctl($pid, "get count");

Examples of Free Mode

Command "get count" allows you to get the current count value of ST.

```
$tick = pid_ioctl($pid, "get count");
```

example of up counter

This example sets ST to up counter and prints counter value in every second.

```
$pid = pid_open("/mmap/st0");               // open ST 0
pid_ioctl($pid, "set mode free");           // set mode: free
pid_ioctl($pid, "set div sec");             // set unit: second
pid_ioctl($pid, "set dir up");              // set direction: up counter
pid_ioctl($pid, "start");                   // start ST
for($i=0; $i<10; $i++)
{
    $value = pid_ioctl($pid, "get count");  // read the count value
    echo "$value₩r₩n";                      // print the count value
    sleep(1);
}
pid_close($pid);
```

example of down counter

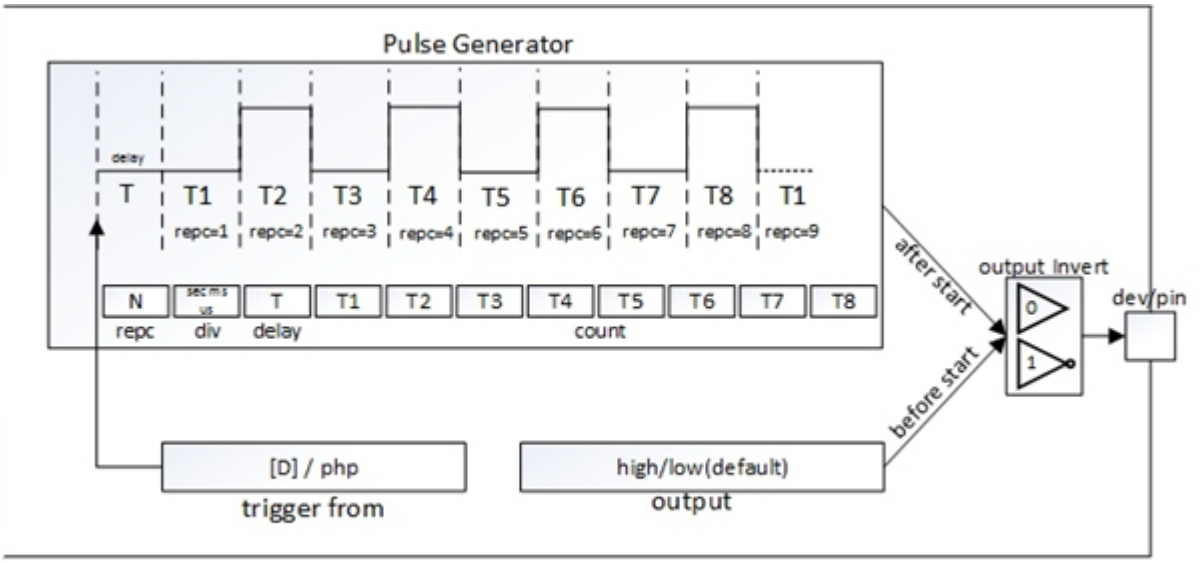
This example sets ST to down counter with the starting count value and prints counter value in every second.

```
$pid = pid_open("/mmap/st0");               // open ST 0
pid_ioctl($pid, "set mode free");           // set mode: free
pid_ioctl($pid, "set div sec");             // set unit: second
pid_ioctl($pid, "set dir down");            // set direction: down counter
pid_ioctl($pid, "set count 10");            // set count value: 10
pid_ioctl($pid, "start");                   // start ST
for($i = 0; $i < 10; $i++)
{
    $value = pid_ioctl($pid, "get count");  // read the count value
```

```
    echo "$value₩r₩n";                    // print the count value
    sleep(1);
}
pid_close($pid);
```

Toggle Mode

This mode toggles the state of specified output pin of ST.



ST(output toggle) Block Diagram

Available commands in toggle mode are as follows:

Command	Sub Command			Description
set	mode	output	toggle	set mode: toggle
	div	sec		set unit: second
		ms		set unit: millisecond
		us		set unit: microsecond
	output	od		open-drain
		pp		push-pull
		low		output LOW
		high		output HIGH
		dev	uio0 #pin	set output device and pin
		invert	0	not invert output
			1	invert output
	count	[T1] ... [T8]		set output timing parameters
	delay	[D]		set delay before output signal
	repc	[N]		set repeat count
	trigger	from	st#	set trigger target: st0 ~ st7
			php	set trigger target: none
reset	-			reset
get	state			get current state
	repc			get remaining repeat count
start	-			start
stop	-			stop

Set Output

Sub commands of "set output" command in toggle mode are as follows:

Sub Command	Syntax
set output pin	pid_ioctl($pid, "set output dev uio0 0"); // pin 0 of uio0

Sub Command	Syntax
open-drain	pid_ioctl($pid, "set output od");
push-pull	pid_ioctl($pid, "set output pp");
output HIGH	pid_ioctl($pid, "set output high");
output LOW	pid_ioctl($pid, "set output low");

invert output	pid_ioctl($pid, "set output invert 1"); // inverted output
	pid_ioctl($pid, "set output invert 0"); // normal output

All output sub commands are implemented right after each command line is executed.

Set Delay

This command is for giving delay before PHPoC outputs signal. The unit of delay depends on the unit which is set by "set div" command.

Command	Syntax
set delay	pid_ioctl($pid, "set delay D");

Set Repeat Count

This command is for setting repeat count of output. You can set any values from zero to 1 billion for the repeat count N. If you do not specify N, it is set to zero by default. Setting this value to zero means the maximum repeat count (1 billion).

Command	Syntax
set repc	pid_ioctl($pid, "set repc N");

Set Count Values

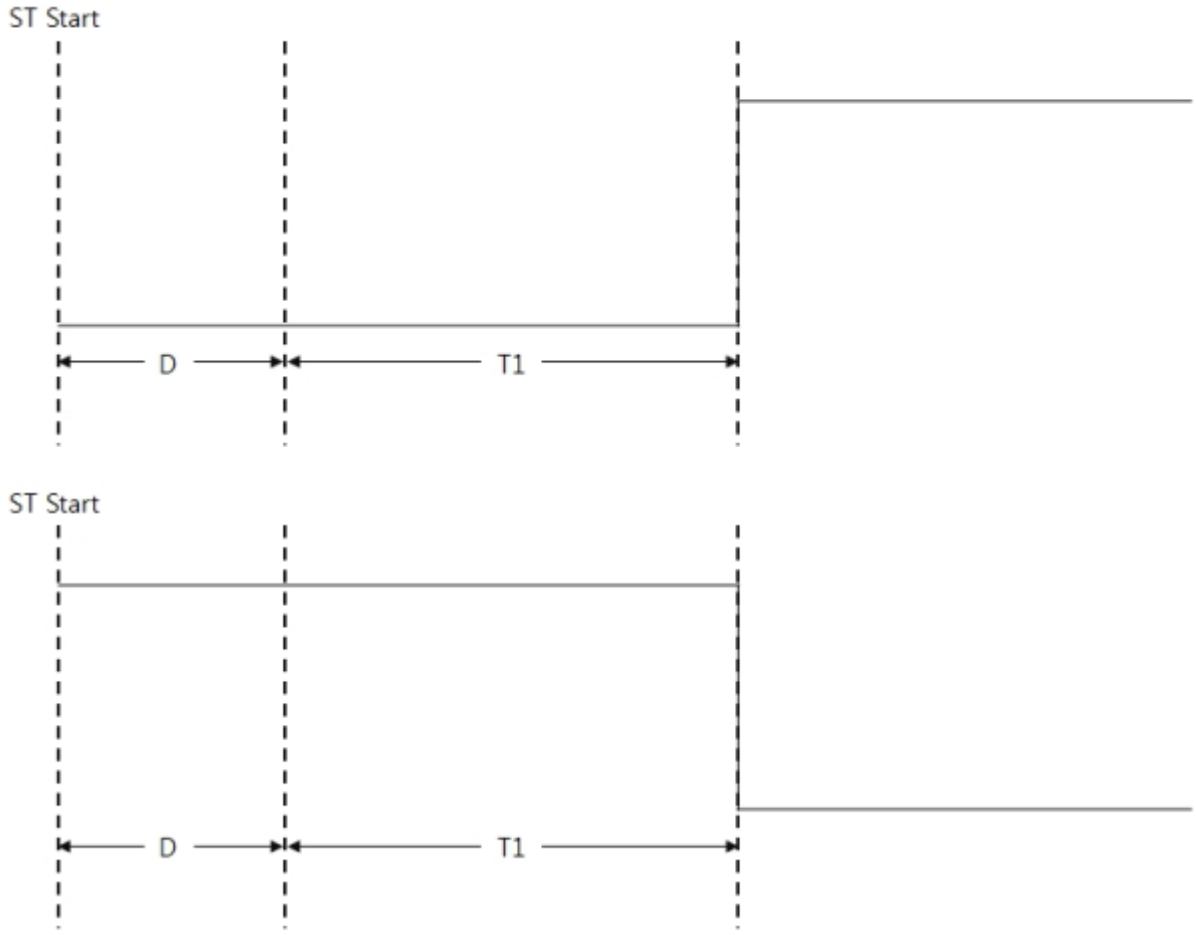
This command is for defining point of time to output signal. In toggle mode, the valid number of count value ranges from one to eight. How to use this command is as follows:

Command	Syntax
set count	pid_ioctl($pid, "set count T1 T2 ... T8");

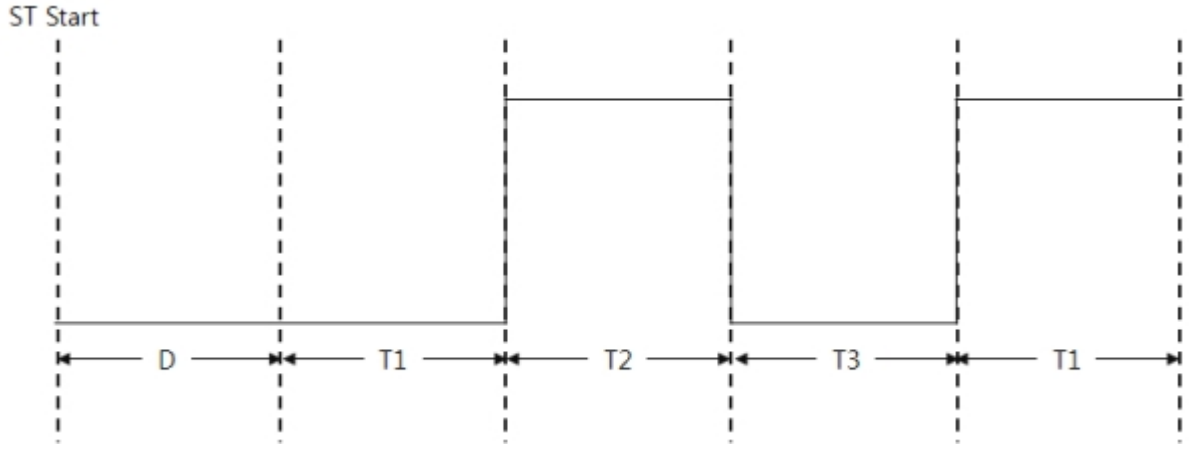
Available values for counts in toggle mode are as follows:

Unit	Available Count Values (10μs ~ half an hour)
microsecond	10 ~ 1,800,000,000
millisecond	1 ~ 1,800,000
second	1 ~ 1,800

The figure below shows waveform in the case of setting just one count value T1 with delay D.



If you set two count values or more than that, every count value is used in order. When repeat count is greater than the number of setting counts, count values are used again from the first count value. For example, waveform of setting 3 count values (T1, T2 and T3) with 4 repeat count including delay D is as follows:



Set Trigger

This command is used when you want to synchronize an ST start time with another ST. Target of trigger should be one of ST devices.

Target	Syntax
ST(st0/1...)	pid_ioctl($pid, "set trigger from st0");
php	pid_ioctl($pid, "set trigger from php");

Default value of trigger target is "php"(no target).

Get Repeat Count

Command "get repc" is for reading the remaining repeat count which will be executed.

Command	Syntax
get repc	pid_ioctl($pid, "get repc");

Example of Toggle Mode

Toggle mode toggles output signals.

example of toggle mode

```
$pid = pid_open("/mmap/st0");                // open ST 0
pid_ioctl($pid, "set div sec");              // set unit: second
pid_ioctl($pid, "set mode output toggle");  // set mode: toggle
pid_ioctl($pid, "set output dev uio0 0");   // set output device / pin: uio0 / 0
pid_ioctl($pid, "set repc 1");               // set repeat count: 1
pid_ioctl($pid, "set count 1");              // set count: T1 only
pid_ioctl($pid, "start");                    // start ST
while(pid_ioctl($pid, "get state"));
pid_close($pid);
```

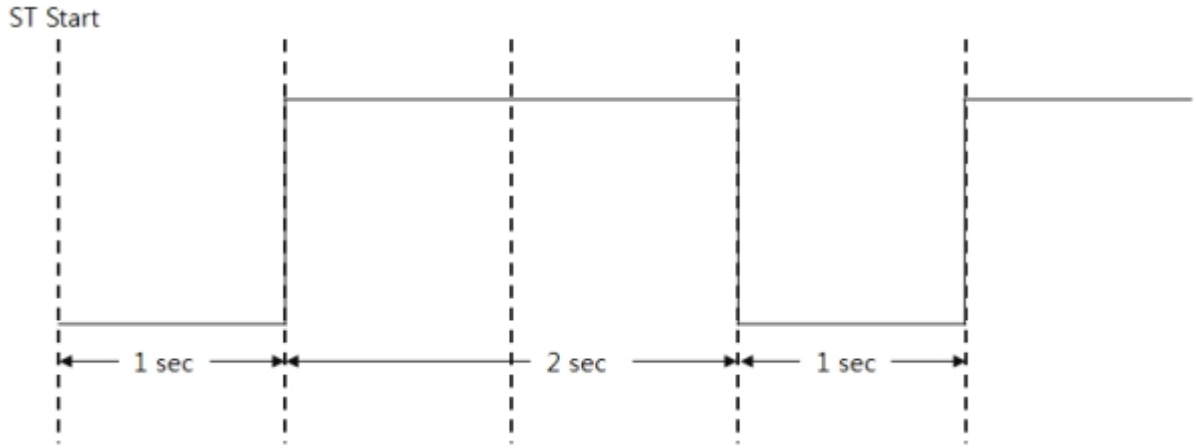
The meaning of "set count" is amount of time between starting ST and output toggle signal. The figure below shows waveform of the above example.

example of repetitive toggle mode

```
$pid = pid_open("/mmap/st0");                // open ST 0
pid_ioctl($pid, "set div sec");              // set unit: second
pid_ioctl($pid, "set mode output toggle");  // set mode: toggle
pid_ioctl($pid, "set output dev uio0 0");   // set output device / pin: uio0 / 0
pid_ioctl($pid, "set repc 3");               // set repeat count: 3
pid_ioctl($pid, "set count 1 2 1");          // set count values: 1, 2 and 1
pid_ioctl($pid, "start");                    // start ST
```

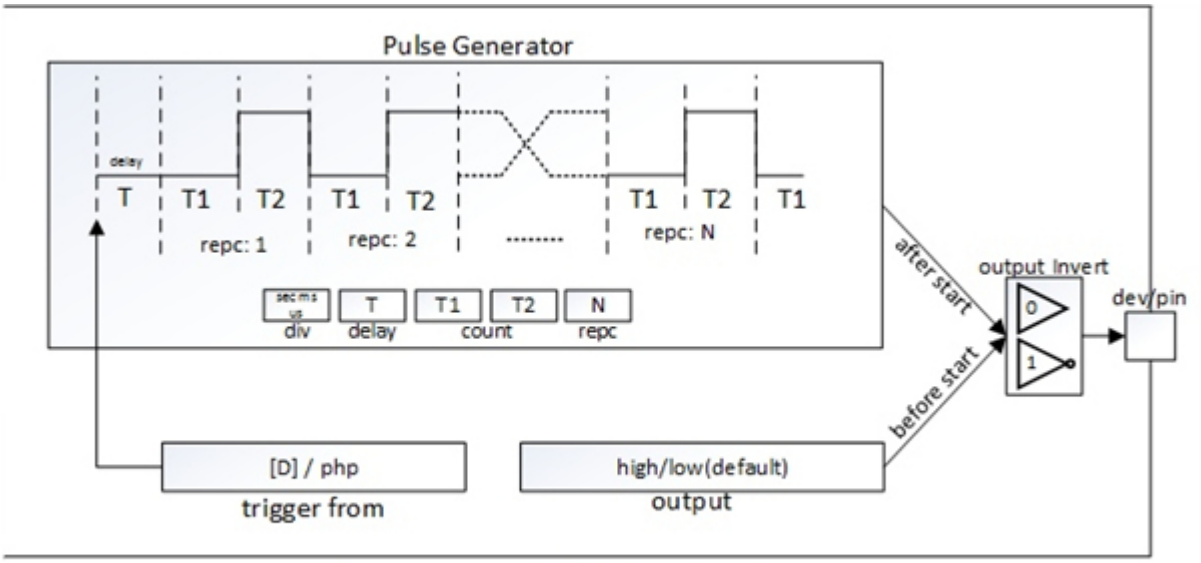
```
while(pid_ioctl($pid, "get state"));
pid_close($pid);
```

In the example above, three count values (T1, T2 and T3) are set and those are 1, 2 and 1 second. The waveform is as follows:



Pulse Mode

Pulse mode outputs square waves.



ST(output pulse) Block Diagram

Available commands in pulse mode are as follows:

Command	Sub Command				Description
set	mode	output	pulse		set mode: pulse
	div	sec			set unit: second
		ms			set unit: millisecond
		us			set unit: microsecond
	output	od			open-drain
		pp			push-pull
		low			output: LOW
		high			output: HIGH
		dev	uio0	#pin	set output device and pin
		invert	0		not invert output
			1		invert output
	count	[T1] [T2]			set output timing parameters
	delay	[D]			set delay
	repc	[N]			set repeat count
	trigger	from	st#		set trigger target: st0 ~ st7
			php		set trigger target: none
reset	-				reset
get	state				get state
	repc				get remaining repeat count
start	-				start
stop	-				stop

Set Output

Sub commands of "set output" command in pulse mode are as follows:

Sub Command	Syntax
set output pin	pid_ioctl($pid, "set output dev uio0 0"); // uio0의 0번 핀

Sub Command	Syntax
open-drain	pid_ioctl($pid, "set output od");
push-pull	pid_ioctl($pid, "set output pp");
output HIGH	pid_ioctl($pid, "set output high");
output LOW	pid_ioctl($pid, "set output low");

invert output	pid_ioctl($pid, "set output invert 1"); // inverted output
	pid_ioctl($pid, "set output invert 0"); // normal output

All commands are implemented right after each command line is executed.

Set Delay

This command is for giving delay before PHPoC outputs signal. The unit of delay depends on the unit which is set by "set div" command.

Command	Syntax
set delay	pid_ioctl($pid, "set delay D");

Set Repeat Count

This command is for setting repeat count of output. You can set any values from zero to 1 billion for the repeat count N. If you do not specify N, it is set to zero which is default value. Setting this value to zero means the maximum repeat count (1 billion).

Command	Syntax
set repc	pid_ioctl($pid, "set repc N");

Set Count Values

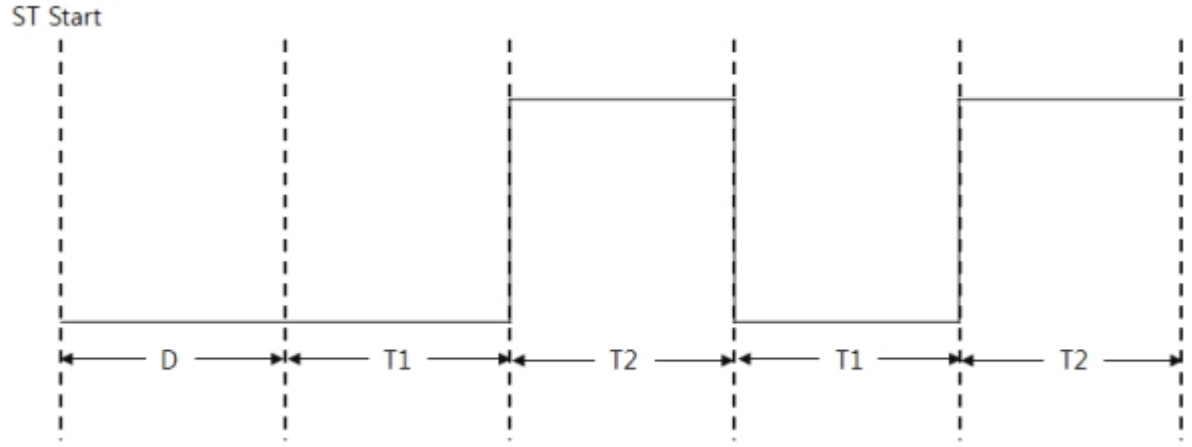
This command is for defining point of time to output signal. In pulse mode, two count values (T1 and T2) are required.

Command	Syntax
set count	pid_ioctl($pid, "set count T1 T2");

Available values for count T1 and T2 in pulse mode are as follows:

Unit	Available Count Values
microsecond	0, 10 ~ 1,800,000,000
millisecond	0 ~ 1,800,000
second	0 ~ 1,800

The figure below shows waveform in the case of setting T1 and T2 with delay D in pulse mode.



Get Repeat Count

Command "get repc" is for reading the remaining repeat count which will be executed.

Command	Syntax
get repc	pid_ioctl($pid, "get repc");

Set Trigger

This command is used when you want to synchronize an ST start time with another ST. Target of trigger should be one of ST devices.

Target	Syntax
ST(st0/1...)	pid_ioctl($pid, "set trigger from st0");
php	pid_ioctl($pid, "set trigger from php");

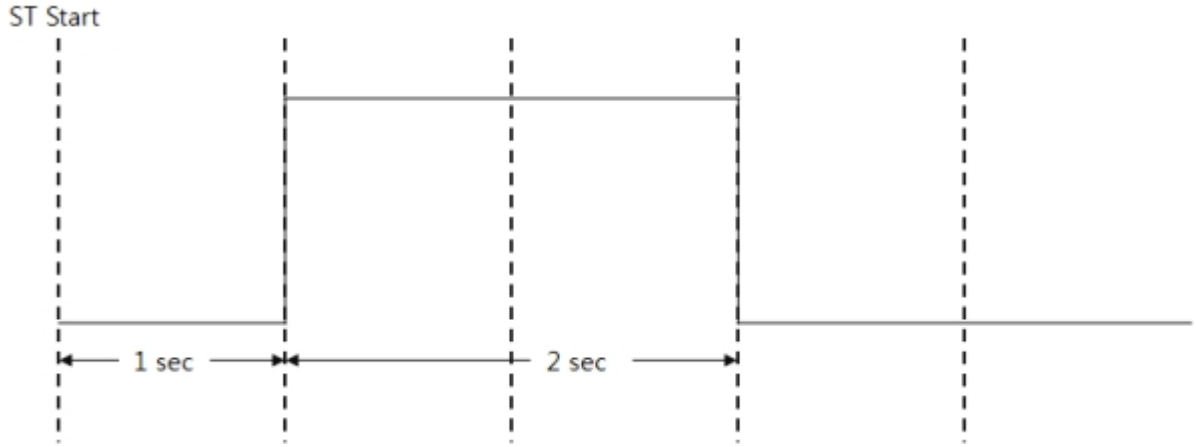
Default value of trigger target is "php"(no target).

Example of Pulse Mode

example of pulse mode (HIGH pulse)

```
$pid = pid_open("/mmap/st0");               // open ST 0
pid_ioctl($pid, "set div sec");             // set unit: second
pid_ioctl($pid, "set mode output pulse");  // set mode: pulse
pid_ioctl($pid, "set output dev uio0 0");  // set output device / pin: uio0 / 0
pid_ioctl($pid, "set count 1 2");           // set count values: 1 and 2
pid_ioctl($pid, "set repc 1");              // set repeat count: 1
pid_ioctl($pid, "start");                   // start ST
while(pid_ioctl($pid, "get state"));
pid_close($pid);
```

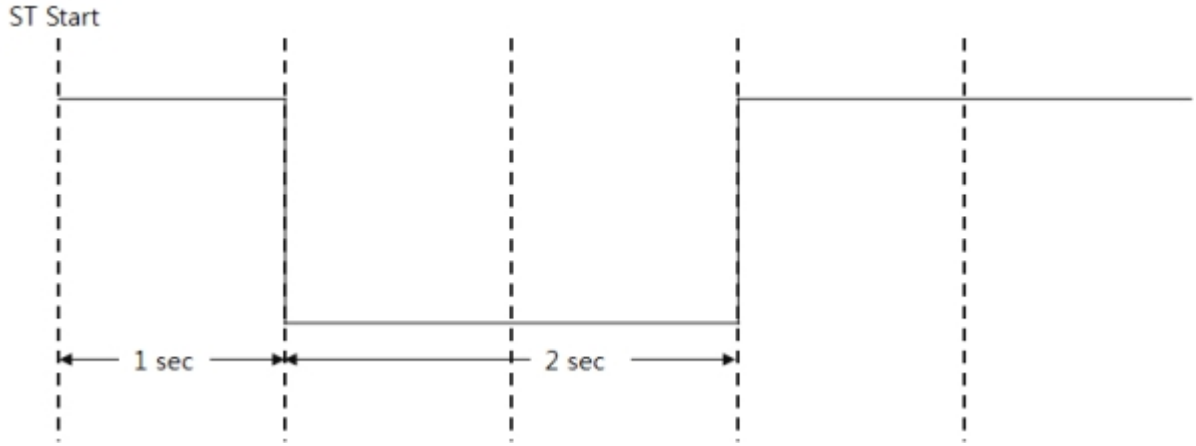
Pulse mode basically changes level from low to high. The timing of change depends on both division rate and count values (T1 and T2). The following figure shows waveform of the example above.



example of pulse mode (LOW pulse)

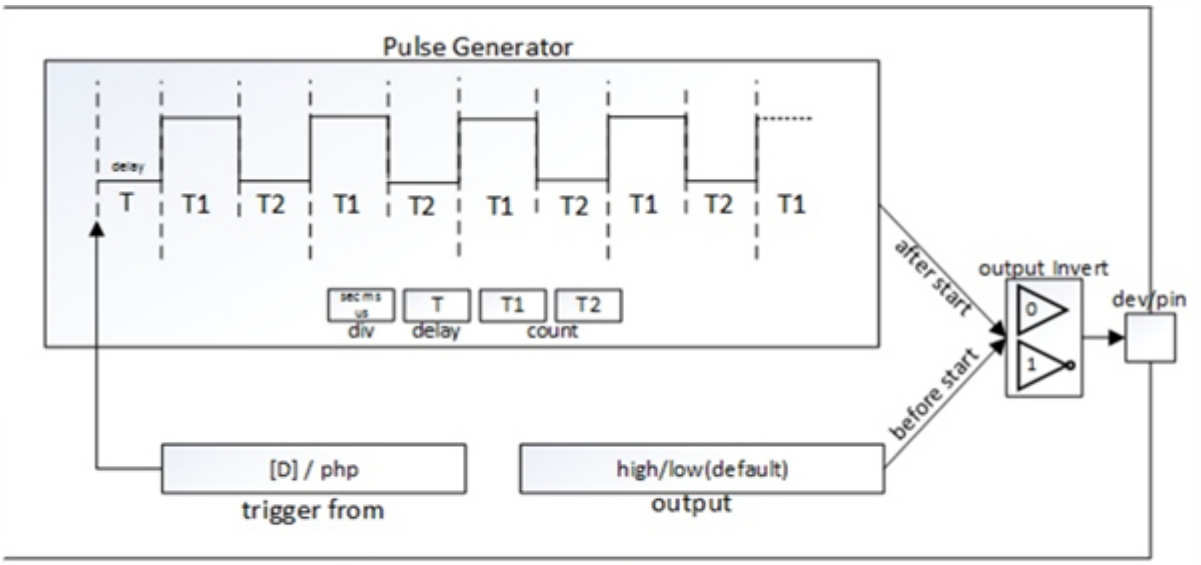
```
$pid = pid_open("/mmap/st0");                 // open ST 0
pid_ioctl($pid, "set div sec");               // set unit: second
pid_ioctl($pid, "set mode output pulse");    // set mode: pulse
pid_ioctl($pid, "set output dev uio0 0");    // set output device / pin: uio0 / 0
pid_ioctl($pid, "set count 1 2");             // set count values: 1 and 2
pid_ioctl($pid, "set output invert 1");      // invert output
pid_ioctl($pid, "set repc 1");                // set repeat count: 1
pid_ioctl($pid, "start");                     // start ST
while(pid_ioctl($pid, "get state"));
pid_close($pid);
```

After executing the command line "set output invert 1", all output levels are inverted including a pulse output. The figure below shows waveform of example above.



PWM Mode

PWM mode is called infinite pulse mode so syntax is almost the same with pulse mode but a little difference.



ST(output pwm) Block Diagram

Available commands in PWM mode are as follows:

Command	Sub Command			Description
set	mode	output	pwm	set mode: PWM
	div	sec		set unit: second
		ms		set unit: millisecond
		us		set unit: microsecond
	output	od		open-drain
		pp		push-pull
		low		output LOW
		high		output HIGH
		dev	uio0 #pin	set output device and pin
		invert	0	not invert output
			1	invert output
	count	[T1] [T2]		set output timing parameters
	delay	[D]		set delay
	trigger	from	st#	set trigger target: st0 ~ st7
			php	set trigger target: none
reset	-			reset
get	state			get current state
start	-			start
stop	-			stop

Set Count Values

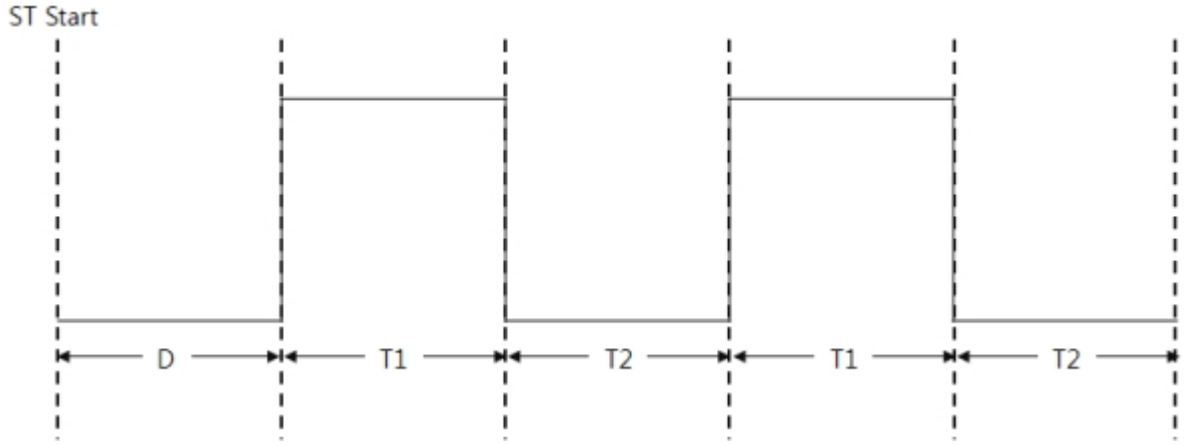
Count values defines the point of time to change levels. In PWM mode, two count values are required. How to set count values is as follows:

Command	Syntax
set count	pid_ioctl($pid, "set count T1 T2");

Available count values in pwm mode are as follows:

Unit	Available Count Values (0 ~ half an hour)
microsecond	0, 10 ~ 1,800,000,000
millisecond	0 ~ 1,800,000
second	0 ~ 1,800

The figure below shows waveform in the case of setting T1 and T2 with delay D in PWM mode.



Set Trigger

This command is used when you want to synchronize an ST start time with another ST. Target of trigger should be one of ST devices.

Target	Syntax
ST(st0/1...)	pid_ioctl($pid, "set trigger from st0");
php	pid_ioctl($pid, "set trigger from php");

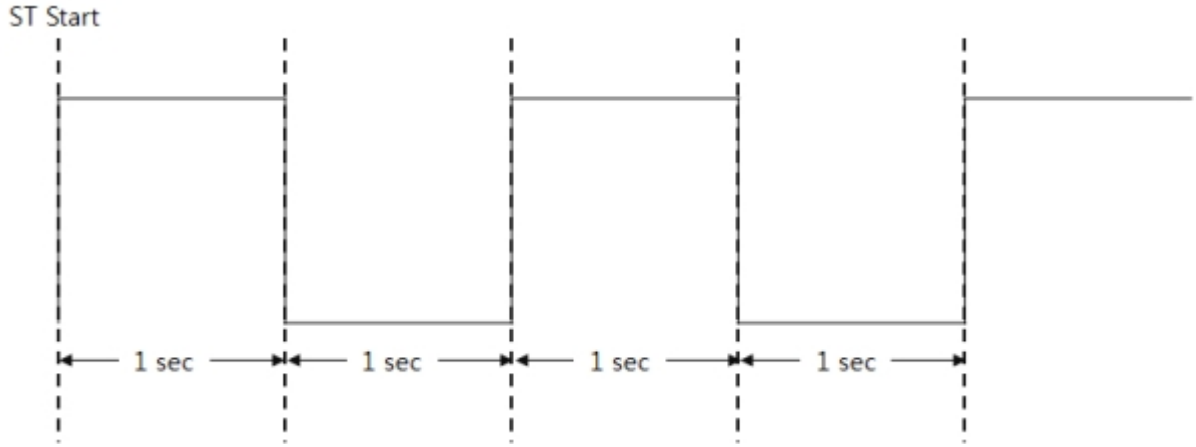
Default value of trigger target is "php"(no target).

Example of PWM Mode

example of PWM mode

```
$pid = pid_open("/mmap/st0");               // open ST 0
pid_ioctl($pid, "set div sec");             // set unit: second
pid_ioctl($pid, "set mode output pwm");     // set mode: PWM
pid_ioctl($pid, "set output dev uio0 0");  // set output dev / pin: uio0 / 0
pid_ioctl($pid, "set count 1 1");           // set count values: 1 and 1
pid_ioctl($pid, "start");                   // start ST
sleep(10);
pid_ioctl($pid, "stop");                    // stop ST
pid_close($pid);
```

The figure below shows waveform of the example above.


ST Start
1 sec
1 sec
1 sec
1 sec

Trigger

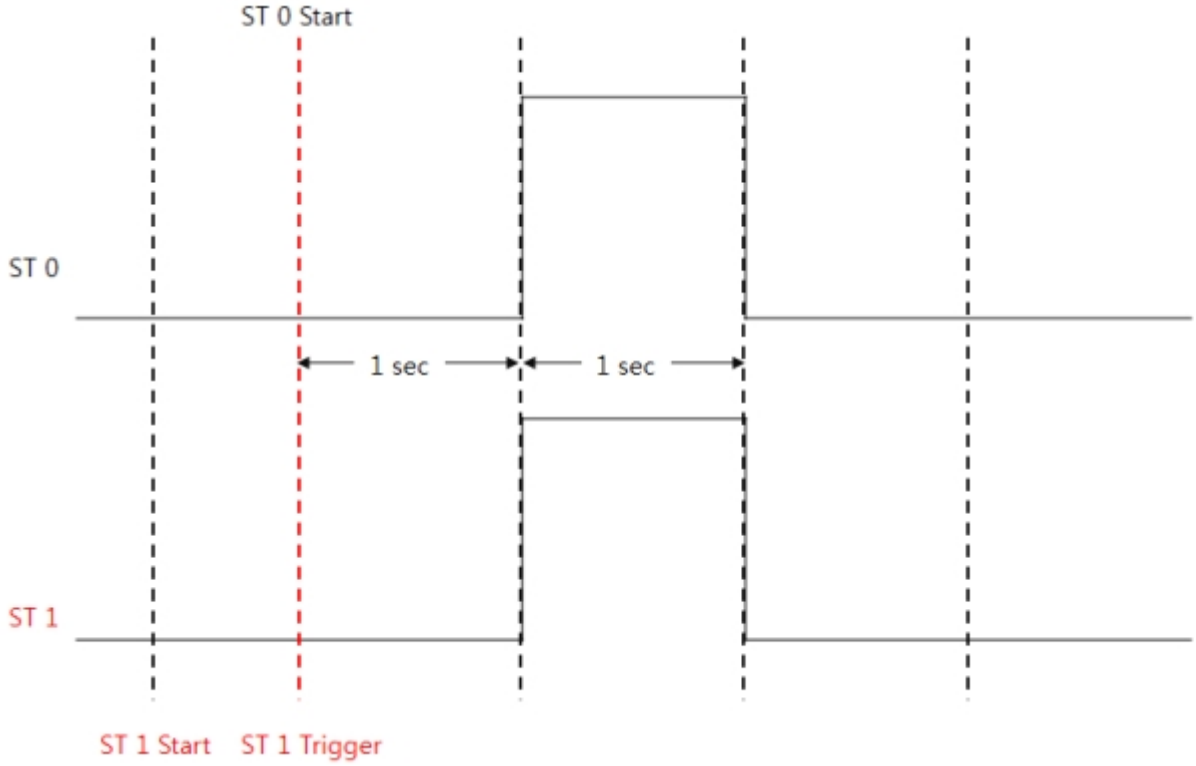
Trigger command is used when you want to synchronize an ST start time with another ST. The example below shows how to synchronize ST1 to ST0 using trigger.

example of trigger

```
$pid0 = pid_open("/mmap/st0");               // open ST 0
pid_ioctl($pid0, "set div sec");             // set unit: second
pid_ioctl($pid0, "set mode output pulse");  // set mode: pulse
pid_ioctl($pid0, "set count 1 1");           // set count values: 1 and 1
pid_ioctl($pid0, "set repc 2");              // set repeat count: 2
pid_ioctl($pid0, "set output dev uio0 0");  // set output dev / pin: uio0 / 0

$pid1 = pid_open("/mmap/st1");               // open ST 1
pid_ioctl($pid1, "set div sec");             // set unit: second
pid_ioctl($pid1, "set mode output pulse");  // set mode: pulse
pid_ioctl($pid1, "set trigger from st0");   // set trigger target: st0
pid_ioctl($pid1, "set count 1 1");           // set count values: 1 and 1
pid_ioctl($pid1, "set repc 2");              // set repeat count: 2
pid_ioctl($pid1, "set output dev uio0 1");  // set output dev / pin: uio0 / 1

pid_ioctl($pid1, "start");                   // start ST 1
pid_ioctl($pid0, "start");                   // start ST 0

while(pid_ioctl($pid1, "get state"));
pid_close($pid0);
pid_close($pid1);
```

As you see the example above, ST which you want to synchronize the output time should start before the trigger target starts.

The output is as follows:



error range of ST

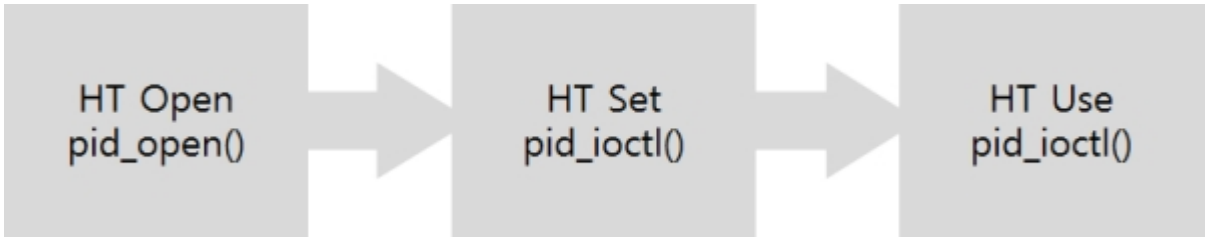
ST leads some error ranges and those are as follows:

Case	Error Range
Simultaneously use 2 STs	approximately 1μs
Simultaneously use 8 STs	approximately 4μs

※ Use HT if you need the high accuracy.

Steps of Using HT

General steps of using HT are as follows:



Opening HT

To open HT, pid_open function is required.

```
$pid = pid_open("/mmap/ht0");       // open HT0
```

※ Refer to Appendix for detailed HT information depending on the types of products.

Setting and Using HT

HT provides four operation modes and you need to use required commands of pid_ioctl function in each mode.

Mode	Description
Output Pulse mode	mode to output pulse signal through an HT pin
Output Toggle mode	mode to output toggle signal through an HT pin
Output PWM mode	mode to output infinite pulse through an HT pin
Capture mode	mode to capture signals from an HT pin

Common Commands

Commands listed in the table below are used in all modes of HT.

Command	Sub Command			Description
set	mode	output	pulse	set mode: output pulse
			toggle	set mode: output toggle
			pwm	set mode: output infinite pulse
		capture	rise	set mode: capture with rising edge
			fall	set mode: capture with falling edge
			toggle	set mode: capture with rising or falling edge
	div	ms		set unit: millisecond
		us		set unit: microsecond
	trigger	from	ht0	set trigger target (should be ht0)
			php	set trigger target (none)
reset	-			reset
get	state			get current state
	div			get division rate
	repc			get remaining repeat count
start	-			start
stop	-			stop

Set Mode

HT provides various output modes as well as capture mode. How to set each mode is as follows:

Mode		Syntax
Output	pulse mode	pid_ioctl($pid, "set mode output pulse");
	toggle mode	pid_ioctl($pid, "set mode output toggle");
	pwm mode	pid_ioctl($pid, "set mode output pwm");
Capture	rising edge	pid_ioctl($pid, "set mode capture rise");
	falling edge	pid_ioctl($pid, "set mode capture fall");
	rising/falling edge	pid_ioctl($pid, "set mode capture toggle");

Set Unit

HT provides millisecond and microsecond unit and the default value is microsecond. How to set each unit is as follows:

Unit	Syntax
millisecond	pid_ioctl($pid, "set div ms");
microsecond	pid_ioctl($pid, "set div us");

□ Set Trigger

How to set trigger is as follows:

Target	Syntax
ht0	pid_ioctl($pid, "set trigger from ht0");
php	pid_ioctl($pid, "set trigger from php"); // No trigger target

HT0 is only option for setting HT as target of trigger in output mode. The default value is "php".

Reset

Command "reset" immediately stops operation of HT and reset.

Command	Syntax
reset	pid_ioctl($pid, "reset");

Get Status

This command is for getting status of HT.

Status	Syntax
current state	pid_ioctl($pid, "get state");
division rate	pid_ioctl($pid, "get div");
remaining count of output / capture	pid_ioctl($pid, "get repc");

Return values of this command are as follows:

Division	Return Value	Description
current state	0	stopped
	1 ~ 3	running
division rate	42	microsecond
	42000	millisecond
remaining count of output / capture	0 ~ 64	-

Start

This command starts HT.

Command	Syntax
start	pid_ioctl($pid, "start");

Stop

Command "stop" immediately stops operation of HT. In output modes, state of output pin keeps unchanged.

Command	Syntax
stop	pid_ioctl($pid, "stop");

Toggle Mode

This mode toggles output signal. Available commands in this mode are as follows:

Command	Sub Command			Description
set	mode	output	toggle	set mode: toggle
	div	ms		set unit: millisecond
		us		set unit: microsecond
	output	od		open-drain
		pp		push-pull
		low		output LOW
		high		output HIGH
		invert	0	not invert output
			1	invert output
	count	[T1] ... [T8]		set output timing parameters
	repc	[N]		set repeat count
	trigger	from	ht0	set trigger target: ht0
			php	set trigger target: none
reset	-			reset
get	state			get current state
	div			get division rate
	repc			get remaining repeat count
start	-			start
stop	-			stop

Set Output

Sub commands of "set output" command in toggle mode are as follows:

Division	Syntax
open-drain	pid_ioctl($pid, "set output od");
push-pull	pid_ioctl($pid, "set output pp");
output HIGH	pid_ioctl($pid, "set output high");
output LOW	pid_ioctl($pid, "set output low");

invert output	pid_ioctl($pid, "set output invert 1"); // inverted output
	pid_ioctl($pid, "set output invert 0"); // normal output

All commands are implemented right after each command line is executed.

Set Repeat Count

This command is for setting repeat count of output. You can set any values from zero to 64 for the repeat count N. If you do not specify N, it is set to zero by default. Setting this value to zero means the maximum repeat count (64).

Command	Syntax
set repc	pid_ioctl($pid, "set repc N");

Set Count Values

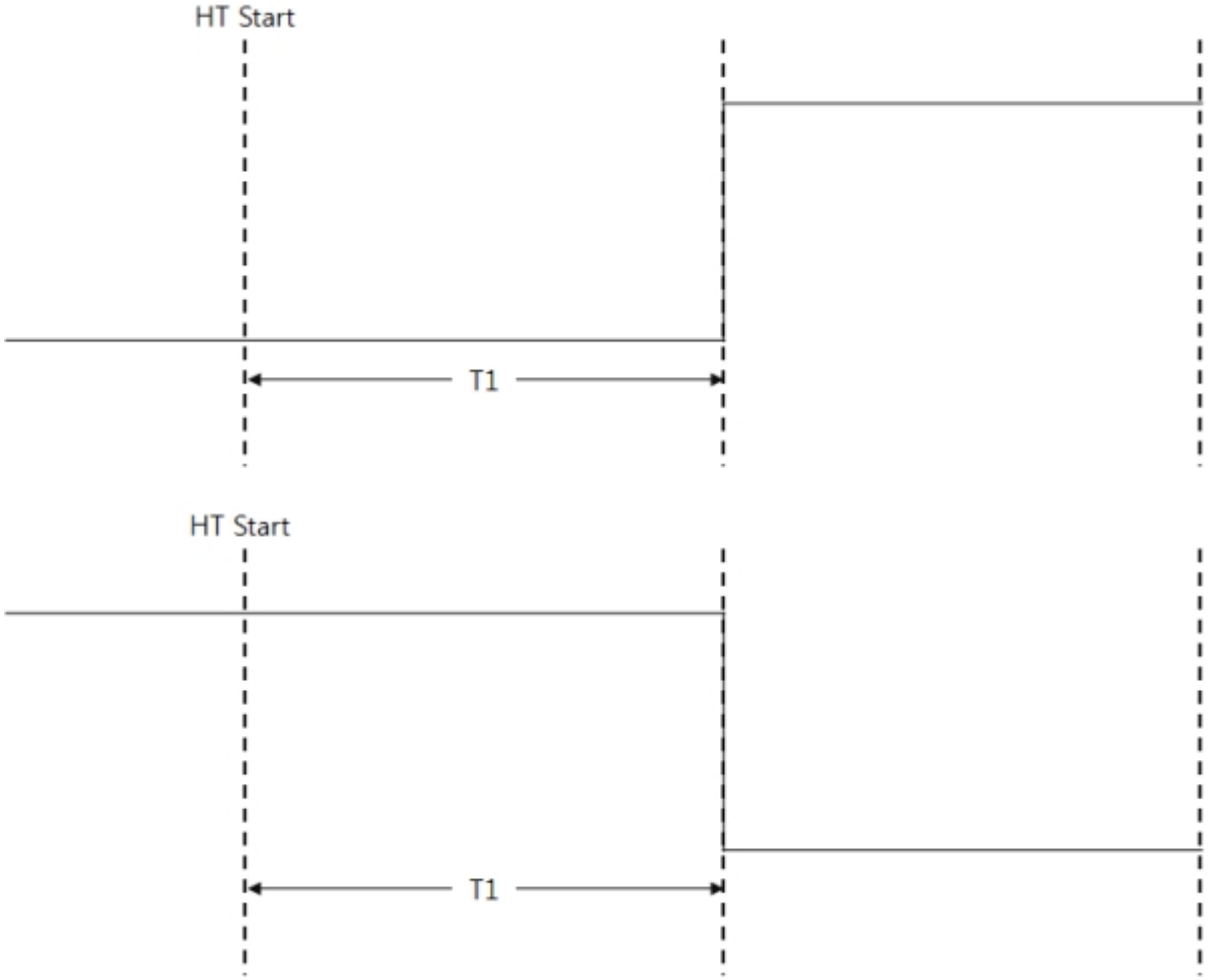
This command is for defining point of time to output signal. In toggle mode, the valid number of count value ranges from one to eight. How to use this command is as follows:

Command	Syntax
set count	pid_ioctl($pid, "set count T1 T2 ... T8");

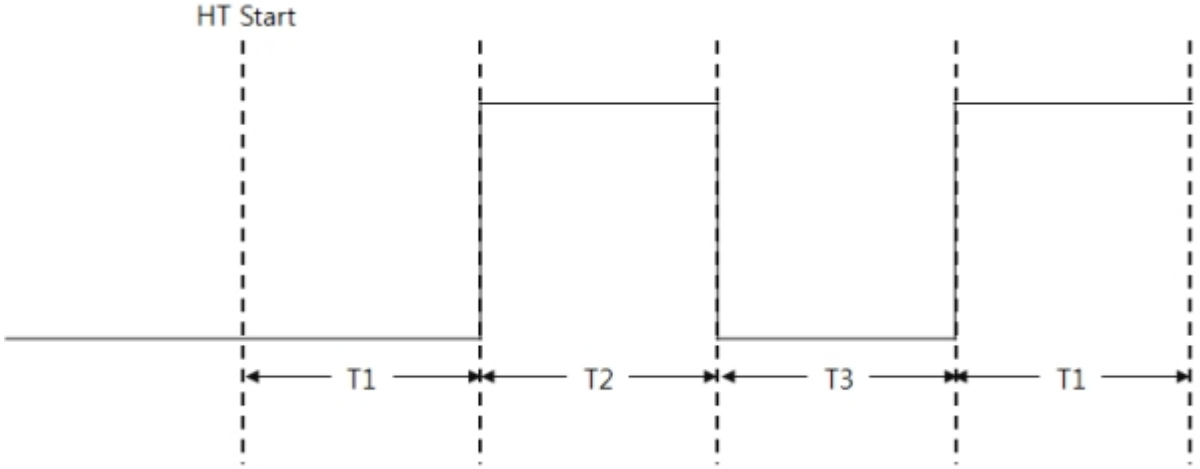
Available values for count in toggle mode are as follows:

Division	Available Count Values
T1 ~ T8	1 ~ 32764

The figure below shows waveform in the case of setting just one count value T1.



If you set two count values or more than that, every count value is used in order. When repeat count is greater than the number of setting counts, count values are used again from the first count value. For example, waveform of setting 3 count values (T1, T2 and T3) with 4 repeat count is as follows:



Example of Toggle Mode

example of toggle mode

```
$pid = pid_open("/mmap/ht0");                // open HT 0
pid_ioctl($pid, "set div us");               // set unit: microsecond
pid_ioctl($pid, "set mode output toggle");  // set mode: toggle
pid_ioctl($pid, "set repc 1");               // set repeat count: 1
pid_ioctl($pid, "set count 1");              // set count T1: 1
pid_ioctl($pid, "start");                    // start HT
while(pid_ioctl($pid, "get state"));
pid_close($pid);
```

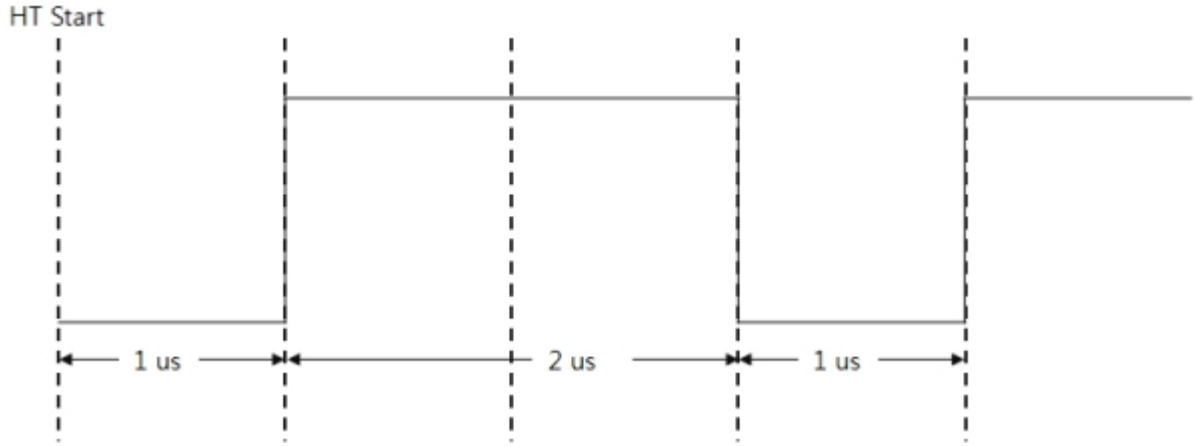
The meaning of "set count" is amount of time between starting HT and output toggle signal. The figure below shows waveform of the above example.

example of repetitive toggle mode

```
$pid = pid_open("/mmap/ht0");                // open HT 0
pid_ioctl($pid, "set div us");               // set unit: microsecond
pid_ioctl($pid, "set mode output toggle");  // set mode: toggle
pid_ioctl($pid, "set repc 3");               // set repeat count: 3
pid_ioctl($pid, "set count 1 2 1");          // set count values: 1, 2 and 1
```

```
pid_ioctl($pid, "start");                    // start HT
while(pid_ioctl($pid, "get state"));
pid_close($pid);
```

In the example above, three count values (T1, T2 and T3) are set and those are 1, 2 and 1 microsecond. The waveform of HT output is as follows:



Pulse Mode

This mode outputs square wave. Available commands in pulse mode are as follows:

Command	Sub Command			Description
set	mode	output	pulse	set mode: pulse
	div	ms		set unit: millisecond
		us		set unit: microsecond
	output	od		open-drain
		pp		push-pull
		low		output: LOW
		high		output: HIGH
		invert	0	not invert output
			1	invert output
	count	[T1] [T2]		set output timing parameters
	repc	[N]		set repeat count
	trigger	from	ht0	set trigger target: ht0
			php	set trigger target: none
reset	-			reset
get	state			get current state
	div			get division rate
	repc			get remaining repeat count
start	-			start
stop	-			stop

Set Output

Sub commands of "set output" command in pulse mode are as follows:

Sub Command	Syntax
open-drain	pid_ioctl($pid, "set output od");
push-pull	pid_ioctl($pid, "set output pp");
output HIGH	pid_ioctl($pid, "set output high");
output LOW	pid_ioctl($pid, "set output low");

invert output	pid_ioctl($pid, "set output invert 1"); // inverted output
	pid_ioctl($pid, "set output invert 0"); // normal output

All commands are implemented right after each command line is executed.

Set Repeat Count

This command is for setting repeat count of output. You can set any values from zero to 64 for the repeat count N. If you do not specify N, it is set to zero which is default value. Setting this value to zero means the maximum repeat count (64).

Command	Syntax
set repc	pid_ioctl($pid, "set repc N");

Set Count Values

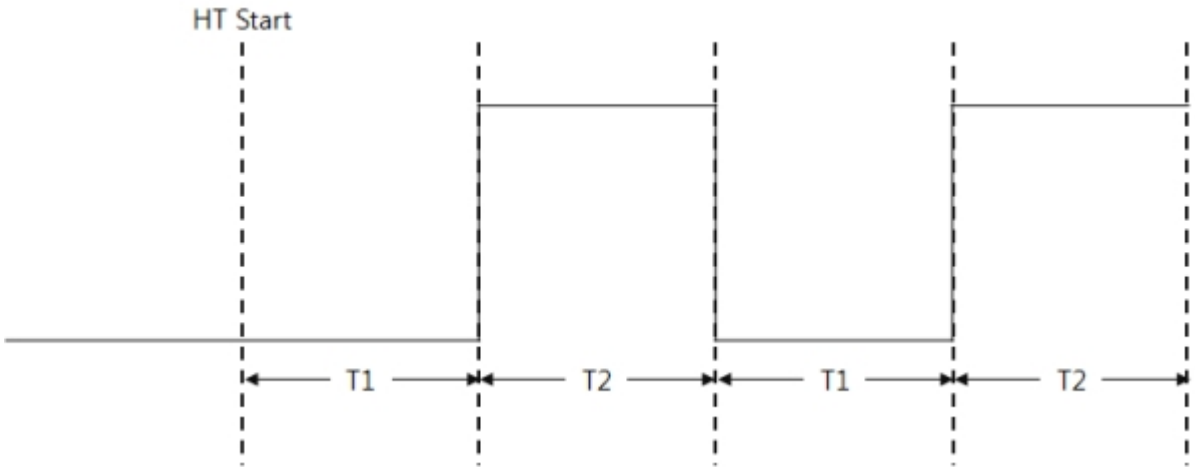
Setting counts is for defining point of time to output signal. In pulse mode, two count values (T1 and T2) are required.

Command	Syntax
set count	pid_ioctl($pid, "set count T1 T2");

Available values for count in pulse mode are as follows:

Division	Available Count Values
T1	1 ~ 32763
T2	1 ~ 32763
T1 + T2	2 ~ 32764

The figure below shows waveform in the case of setting T1 and T2 in pulse mode.

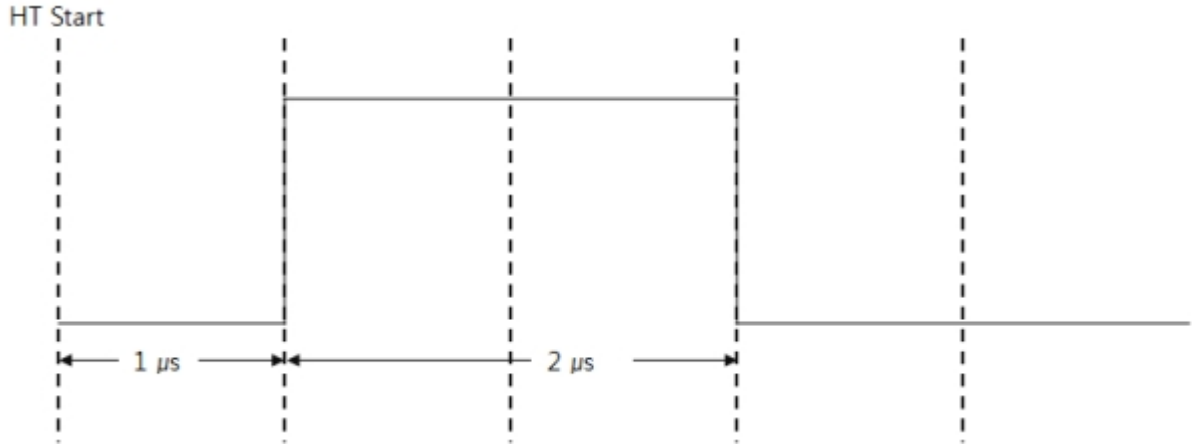


Example of Pulse Mode

example of pulse mode (HIGH pulse)

```
$pid = pid_open("/mmap/ht0");                // open HT 0
pid_ioctl($pid, "set div us");               // set unit: microsecond
pid_ioctl($pid, "set mode output pulse");    // set mode: pulse
pid_ioctl($pid, "set count 1 2");            // set count values: 1, 2
pid_ioctl($pid, "set repc 1");               // set repeat count: 1
pid_ioctl($pid, "start");                    // start HT
while(pid_ioctl($pid, "get state"));
pid_close($pid);
```

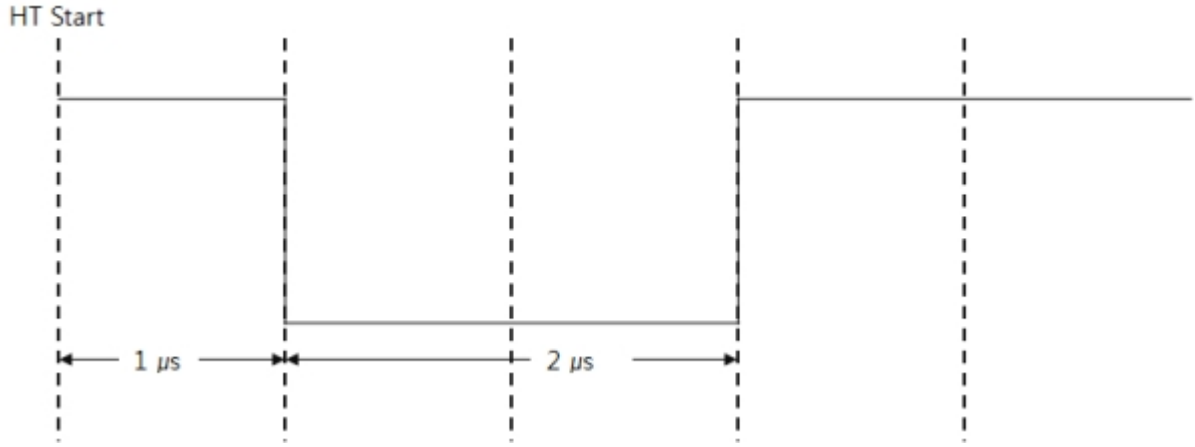
Pulse mode basically changes level from low to high. The timing of change depends on both division rate and count values (T1 and T2). The figure below shows waveform of the example above.



example of pulse mode (LOW pulse)

```
$pid = pid_open("/mmap/ht0");                 // open HT 0
pid_ioctl($pid, "set div us");                // set unit: microsecond
pid_ioctl($pid, "set mode output pulse");    // set mode: pulse
pid_ioctl($pid, "set count 1 2");             // set count values: 1, 2
pid_ioctl($pid, "set repc 1");                // set repeat count: 1
pid_ioctl($pid, "set output invert 1");      // invert output
pid_ioctl($pid, "start");                     // start HT
while(pid_ioctl($pid, "get state"));
pid_close($pid);
```

After executing the command line "set output invert 1", all output levels are inverted including a pulse output. The figure below shows waveform of the example above.



PWM Mode

PWM mode is called infinite pulse mode so syntax is almost the same with pulse mode but a little difference. Available commands in PWM mode are as follows:

Command	Sub Command			Description
set	mode	output	pwm	set mode: pwm
	div	ms		set unit: millisecond
		us		set unit: microsecond
	output	od		open-drain
		pp		push-pull
		low		output LOW
		high		output HIGH
		invert	0	not invert output
			1	invert output
	count	[T1] [T2]		set output timing parameters
	trigger	from	ht0	set trigger target: ht0
			php	set trigger target: none
reset	-			reset
get	state			get current state
	div			get division rate
start	-			start
stop	-			stop

Set Count Values

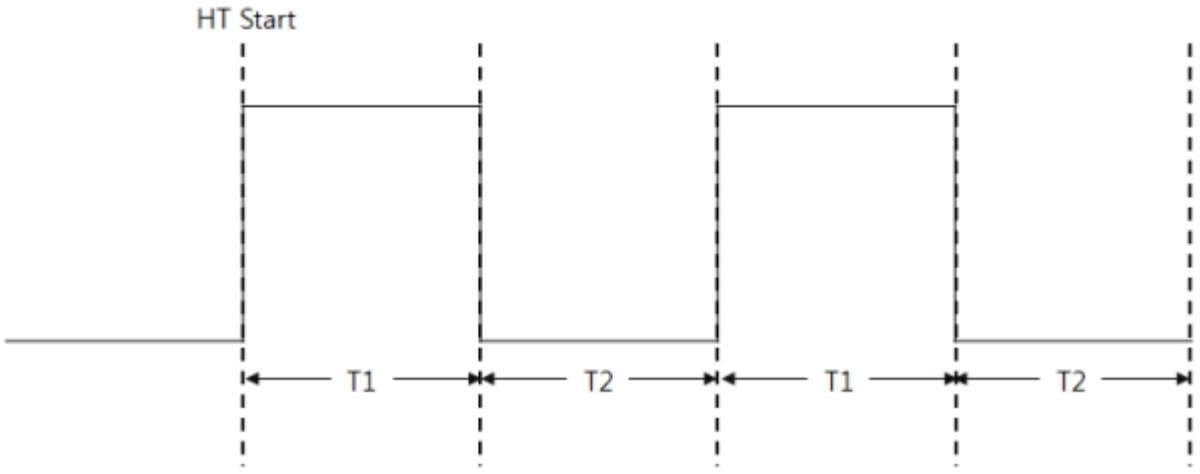
Setting counts is for defining point of time to change levels. In PWM mode, two count values are required. How to set count values is as follows:

Command	Syntax
set count	pid_ioctl($pid, "set count T1 T2");

Available values for count T1 and T2 in PWM mode are as follows:

Division	Available Count Values
T1	0 ~ 32764
T2	0 ~ 32764
T1 + T2	1 ~ 32764

The figure below shows waveform of PWM mode.

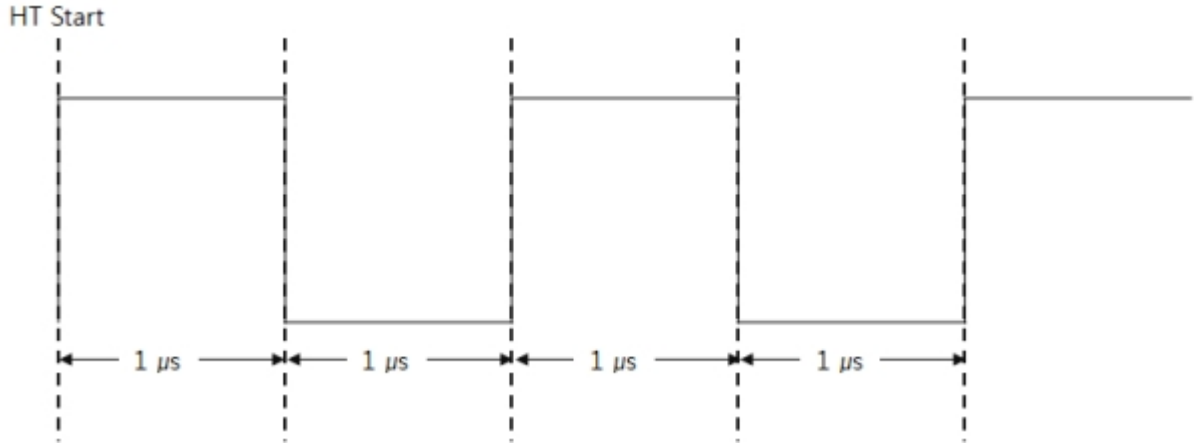


Example of PWM Mode

example of PWM mode

```
$pid = pid_open("/mmap/ht0");             // open HT 0
pid_ioctl($pid, "set div us");            // set unit: microsecond
pid_ioctl($pid, "set mode output pwm"); // set mode: PWM mode
pid_ioctl($pid, "set count 1 1");         // set count values: 1, 1
pid_ioctl($pid, "start");                 // start HT
usleep(50);
pid_ioctl($pid, "stop");                  // stop HT
pid_close($pid);
```

The figure below shows waveform of the example above.



Trigger with Output Modes

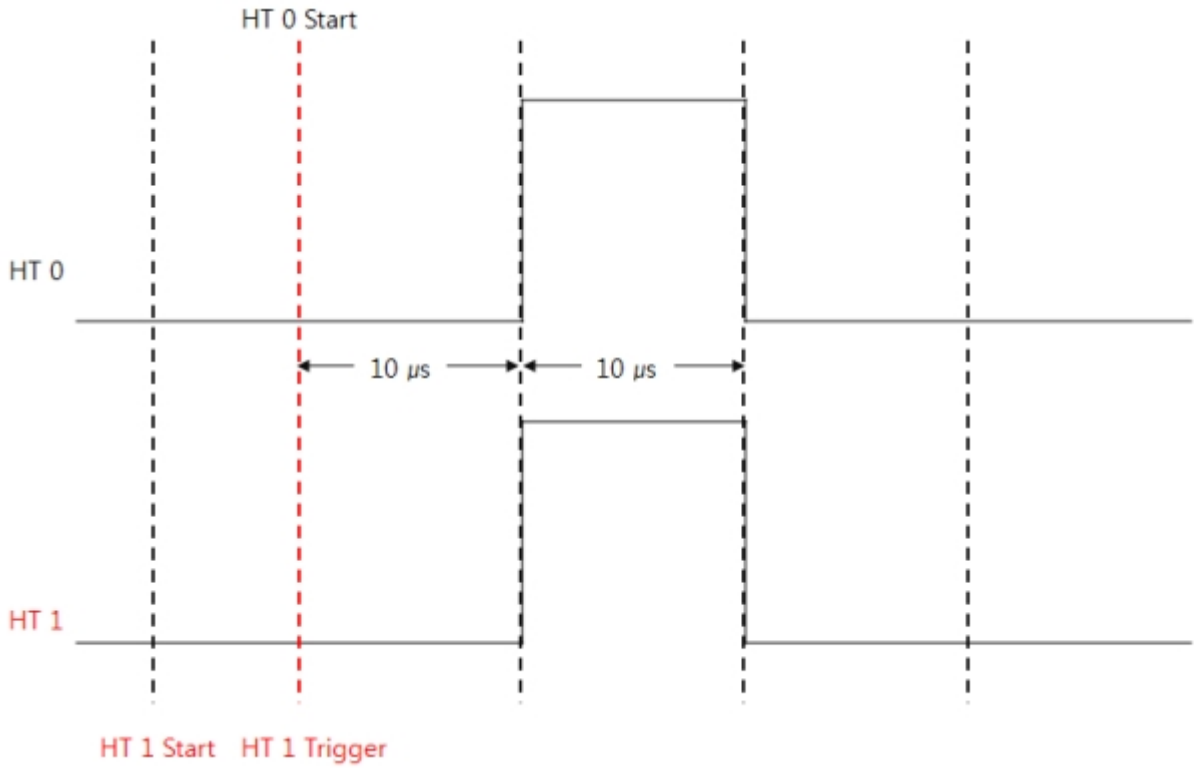
Trigger command is used when you want to synchronize an HT start time with HT 0 in output modes. The example below shows how to synchronize HT 1 to HT 0 using trigger.

example of trigger in pulse mode

```
$pid0 = pid_open("/mmap/ht0");               // open HT 0
pid_ioctl($pid0, "set div us");              // set unit: microsecond
pid_ioctl($pid0, "set mode output pulse");  // set mode: pulse
pid_ioctl($pid0, "set count 10 10");         // set count values: 10 and 10
pid_ioctl($pid0, "set repc 2");              // set repeat count: 2

$pid1 = pid_open("/mmap/ht1");               // open HT 1
pid_ioctl($pid1, "set div us");              // set unit: microsecond
pid_ioctl($pid1, "set mode output pulse");  // set mode: pulse
pid_ioctl($pid1, "set trigger from ht0");   // set a target of trigger: ht0
pid_ioctl($pid1, "set count 10 10");         // set count values: 10 and 10
pid_ioctl($pid1, "set repc 1");              // set repeat count: 1

pid_ioctl($pid1, "start");                   // start HT 1
pid_ioctl($pid0, "start");                   // start HT 0

while(pid_ioctl($pid1, "get state"));
pid_close($pid0);
pid_close($pid1);
```

As you see the example above, HT which you want to synchronize the output time should start before the trigger target (HT 0) starts.

The output signal is as follows:


HT 0 Start
HT 0
10 μs
10 μs
HT 1
HT 1 Start
HT 1 Trigger

Capture Mode

This mode is used when you want to get a count value from a specific time which an event occurs. Available commands in the capture mode are as follows:

Command	Sub Command				Description
set	mode	capture	rise		set capture mode: rising edge
			fall		set capture mode: falling edge
			toggle		set capture mode: rising or falling edge
	div	ms			set unit: millisecond
		us			set unit: microsecond
	repc	[N]			set capture count
	trigger	from	ht0		set a target of trigger: ht0
			php		set a target of trigger: none
			pin	rise	set a type of pin trigger event: rising
				fall	set a type of pin trigger event: falling
				toggle	set a type of pin trigger event: rising and falling
reset	-				reset
get	count	[N]			get a count value
	state				get current state
	repc				get remaining repeat count
start	-				start
stop	-				stop

Set Repeat Count

Repeat count in the capture mode means the number of capturing a specific event. Available repeat count N is from 0 to 64. The default value is 0 and it is regarded as maximum value (64).

Command	Syntax
repeat count	pid_ioctl($pid, "set repc N");

Set Trigger

You can set a trigger target to a HT pin with event as well as HT 0 in the capture mode. How to set this command is as follows:

Division	Syntax
ht0	pid_ioctl($pid, "set trigger from ht0");
pin event	pid_ioctl($pid, "set trigger from pin"); pid_ioctl($pid, "set trigger from pin rise"); pid_ioctl($pid, "set trigger from pin fall"); pid_ioctl($pid, "set trigger from pin toggle");
php	pid_ioctl($pid, "set trigger from php");

There are three event types for pin event: rising edge, falling edge and toggle. You can set the pin event trigger to one of them and the default is toggle when you do not specify any of event types. The default value of setting trigger is "php" which means no trigger target is selected. Although a trigger target is set, you can cancel the designation by specifying the trigger target to "php". In this case, HT starts capturing when it starts.

※ Note that HT2 does not support pin event trigger.

Get a Count Value

This command is for reading a captured count value of HT. Index of the count values can be specified behind this command. How to use it is as follows:

Division	Syntax
[N]th count value	pid_ioctl($pid, "get count [N]");

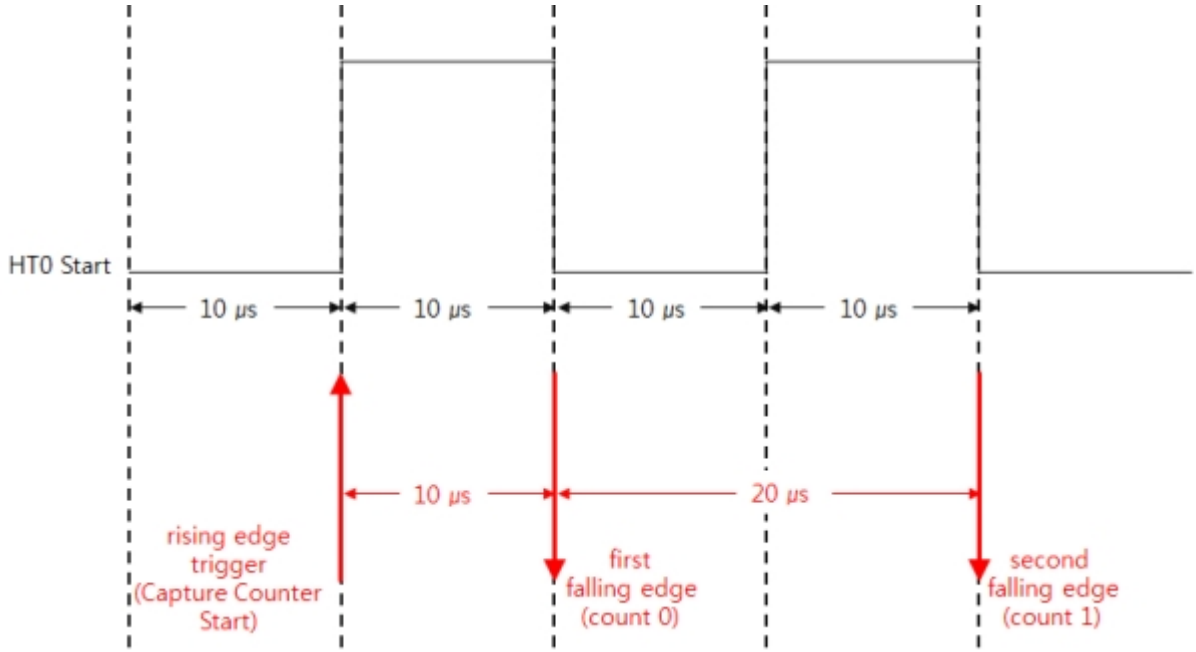
The counter index starts from 0. If you do not specify the index, it will be set to 0. The maximum value of the index is 64.
Accumulated counter values cannot be greater than 32764 in the capture mode. If it exceeds the limitation, HT immediately stops capturing.

Trigger with Capture Mode

Capture mode also defines trigger timing by setting trigger target. Pin input as well as HT 0 can be a trigger target. Trigger timing means the point of time that HT starts internal counter for capturing signal so it is possible to implement capturing signals even if HT is not triggered and the count values are all zero.
The following example shows setting a trigger target to rising edge event and capturing two count values with falling edge.

```
$pid = pid_open("/mmap/ht0");                   // open HT 0
pid_ioctl($pid, "set div us");                  // set unit: microsecond
pid_ioctl($pid, "set mode capture fall");       // set mode: capture with falling edge
// set trigger target: pin event with rising edge
pid_ioctl($pid, "set trigger from pin rise");
pid_ioctl($pid, "set repc 2");                  // set repeat count: 2
pid_ioctl($pid, "start");                       // start HT 0
while(pid_ioctl($pid, "get state"))
    ;
for($i = 0; $i < 2; $i++)
    echo "[$i]", pid_ioctl($pid, "get count $i"), "₩r₩n"; // read count values
pid_close($pid);
```

If two square waves with period of 20μs and duty cycle of 50% are coming while running an example above, counter values of index 0 and 1 are as follows:



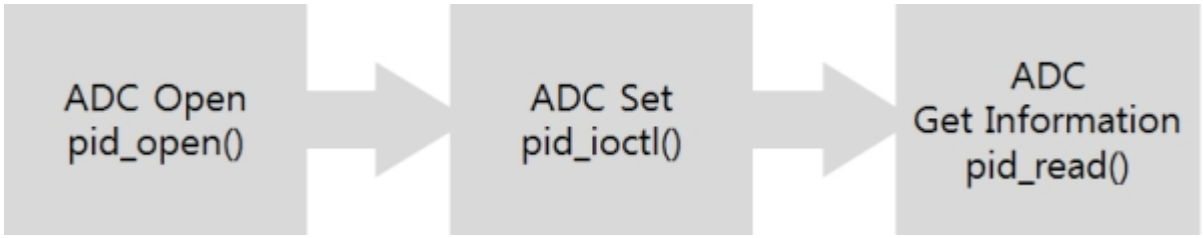
The result is as follows:

```
[0]10
```

```
[1]20
```

Steps of Using ADC

General steps of using ADC are as follows:



Opening ADC

To open ADC, pid_open function is required.

```
$pid = pid_open("/mmap/adc0");        // open ADC
```

※ Refer to Appendix for detailed ADC information depending on the types of products.

Setting ADC Channel

Setting an ADC channel is required to be set before using ADC. If you do not set this, a channel which has the same index of channel with the ADC device's index will be assigned. For example, Channel 0 is automatically assigned to ADC 0 as default channel. You can read ADC values from the channels sequentially by switching among channels.
To set channel or switch to another channel, use the following command:

```
pid_ioctl($pid, "set ch N");      // set channel
```

You can get current channel id by using following command:

```
pid_ioctl($pid, "get ch");        // get the current channel
```

Parameter N means the number of channel.

example of setting ADC channel

```
$pid = pid_open("/mmap/adc0");     // open ADC 0
pid_ioctl($pid, "set ch 1");       // set channel to 1
pid_ioctl($pid, "set ch 2");       // set channel to 2
echo pid_ioctl($pid, "get ch");    // print the current channel(output: 2)
pid_close($pid);                   // close ADC
```

Reading ADC Value

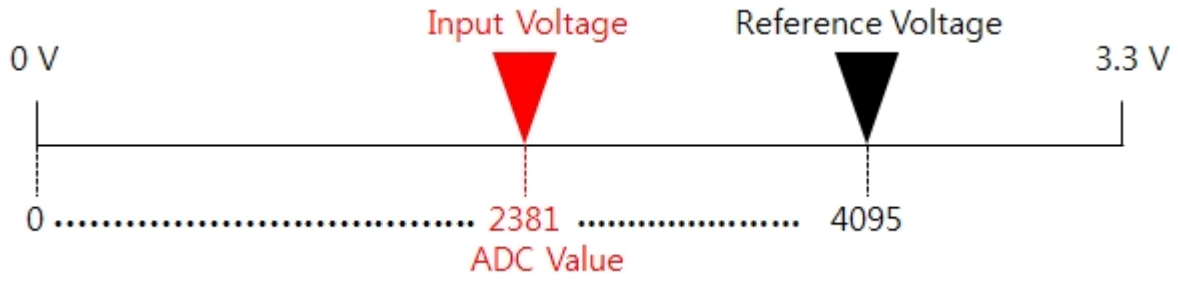
pid_read function is used to read value of ADC.

```
pid_read($pid, $value);
```

Parameter $value is a variable to contain the returned ADC value.

The maximum value of reference voltage is 3.3V and this is used as a default value. If you want to use lower voltage than this, you can input it through reference voltage interface pin(AREF).

An analog input which ranges from 0V to the reference voltage is linearly converted to a digital value which ranges from 0 to 4095.



example of reading ADC value

The example below reads ADC value and calculates the equivalent analog input voltage

```
$adc_value = 0;
$pid = pid_open("/mmap/adc0");          // open ADC 0
pid_ioctl($pid, "set ch 0");            // set channel to 0
pid_read($pid, $adc_value);             // read the ADC value
echo "adc value: $adc_value₩r₩n";       // print the ADC value
$voltage = $adc_value * 3.3 / 4095.0;
echo "voltage : $voltage[V]₩r₩n";       // print the voltage
pid_close($pid);                        // close ADC
```

SPI Overview

PHPoC provides Serial Peripheral Interface (SPI) as one of methods to communicate with other serial device.

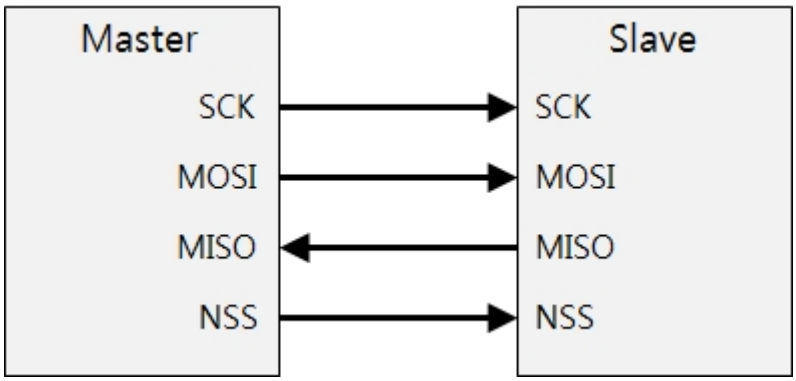
SPI Connection

SPI requires 4 signal lines between a master and a slave.

SPI Signal Lines

Label	Name	Description
SCK	Serial Clock	clock for synchronization
MOSI	Master Output, Slave Input	master's transmission line
MISO	Master Input, Slave Output	slave's transmission line
SS	Slave Select	master's slave select line

SPI connection



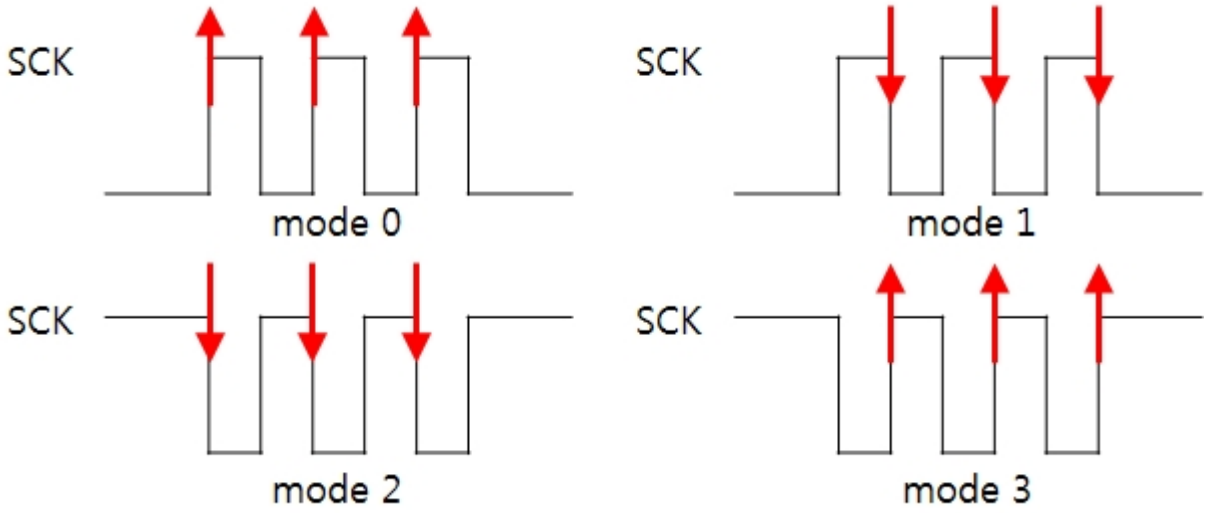
Selecting a Slave

A master's three lines except for SS are commonly connected to all slaves while SS is separately connected to individual slave. Therefore, a master which has three slaves should have at least three SS ports. To select a slave, the master outputs LOW to the line which is connected to the slave and output HIGH to the other lines. That means a master can communicate with a single slave at a time. After finishing communication, the master outputs HIGH to the line.

Data Communication

SPI Modes

There are four SPI modes (0 ~ 3) according to the sampling methods.

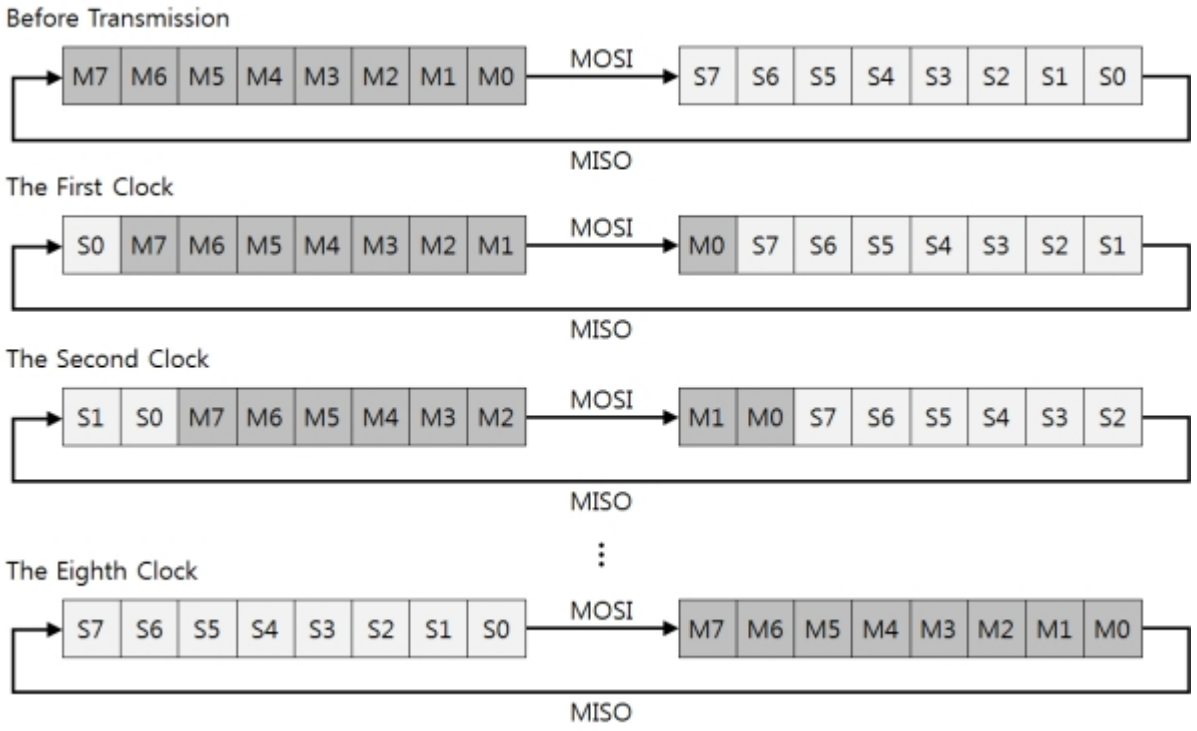


Bit Transmission Order

A master and a slave should be set to have the same bit transmission order. There are two ways: One is to send LSB first and the other is to send MSB first.

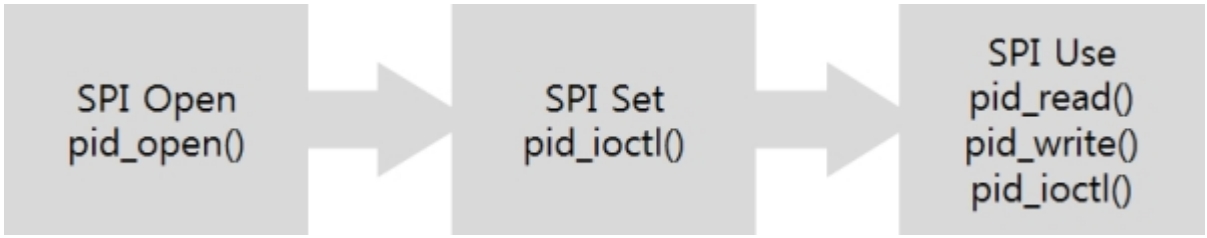
Transmission Sequence

Data buffer of a master and a slave has a type of ring buffer. Thus, sending and receiving data are simultaneously implemented at all times. The following figure shows data flow in case of sending LSB first.



Steps of Using SPI

General steps of using SPI are as follows:



Opening SPI

To open SPI, pid_open function is required.

```
$pid = pid_open("/mmap/spi0");        // open SPI
```

※ Refer to Appendix for detailed SPI information.

Setting and Using SPI

It is required to call pid_ioctl function when you set or use SPI. Available commands of pid_ioctl function for SPI are as follows:

Command	Sub Command		Description
set	lsb	0	set a bit transmission order: MSB first
		1	set a bit transmission order: LSB first
	data	8	set a size of data unit: 8 bits
		16	set a size of data unit: 16 bits
	div	[N]	set a division: 2/4/8/16/32/64/128/256
	mode	[M]	set an SPI mode: 0/1/2/3
get	rxlen		get the number of pending bytes in receive buffer
	txlen		get the number of pending bytes in send buffer
req	start		request to write data
	reset		request to reset bus

Setting SPI

You can set SPI mode, bit transmission order, size of data unit and division by using a "set" command.

Set a Bit Transmission Order

You can set 0 or 1 for setting SPI bit transmission order and the default value is 0. If this value is set to 0, MSB will be transmitted first. How to set the bit transmission order is as follows:

Bit Transmission Order	Syntax
MSB > LSB	pid_ioctl($pid, "set lsb 0");
LSB > MSB	pid_ioctl($pid, "set lsb 1");

Set a Size of Data Unit

You can set 8 or 16 for setting size of data unit and the default value is 8. How to set the size of data unit is as follows:

Size of Data Unit	Syntax
8 bits	pid_ioctl($pid, "set data 8");
16 bits	pid_ioctl($pid, "set data 16");

Set a Data Rate

Data rate depends on division rate of PHPoC's basic clock and you can choose one of the division rate values of 2 / 4 / 8 / 16 / 32 / 64 / 128 / 256. The default value is 256. How to set the data rate is as follows:

Division Rate	Syntax
1 of N	pid_ioctl($pid, "set div N");

※ Basic clock of P4S-341/P4S-342 is 42MHz. Thus, data rate is approximately 164Kbps when the division rate is set to 256.

SPI Mode

You can set one of SPI modes of 0 to 3 and the default value is 3. How to set SPI mode is as follows:

SPI Mode	Syntax
mode 0	pid_ioctl($pid, "set mode 0");
mode 1	pid_ioctl($pid, "set mode 1");
mode 2	pid_ioctl($pid, "set mode 2");
mode 3	pid_ioctl($pid, "set mode 3");

Getting SPI Status

You can get status of SPI by using "get" command.

Get the Number of Pending Bytes in Receive and Send Buffer

How to get the number of pending bytes in receive and send buffer as follows:

Division	Syntax
Send buffer	pid_ioctl($pid, "get txlen");
Receive buffer	pid_ioctl($pid, "get rxlen");

Using SPI

Request to Write Data

You can request to write data with this command. You should input data to send buffer before using this command. After writing data, you should read data from an SPI slave as much as you sent. How to use this command is as follows:

Command	Syntax
Request to Write Data	pid_ioctl($pid, "req start");

Request to Reset Bus

You can reset an SPI bus with this command when communication is bad.

Command	Syntax
Request to Request Bus	pid_ioctl($pid, "req reset");

Example of Using SPI

Write Data to a Slave

The following code is a common example that an SPI master writes data to a slave.

example of writing data

```
$wbuf = 0xA2;                       // Data to be sent
$rbuf = "";

$pid = pid_open("/mmap/spi0");      // open SPI0
pid_ioctl($pid, "set mode 3");      // set SPI mode to 3
pid_ioctl($pid, "set lsb 0");       // set bit transmission order: MSB first
pid_write($pid, $wbuf, 1);          // write 1 byte to buffer: 0xA2
pid_ioctl($pid, "req start");       // request to write data
while(pid_ioctl($pid, "get txlen")) // check the size of transmitted data
    ;
pid_read($pid, $rbuf, 1);           // read 1 byte
pid_close($pid);
```

The reason of reading 1 byte in the bottom of the above example is because reading and writing data are simultaneously implemented at all times in SPI communication.

Read Data from a Slave

The following code is a common example that an SPI master reads data from a slave.

example of reading data

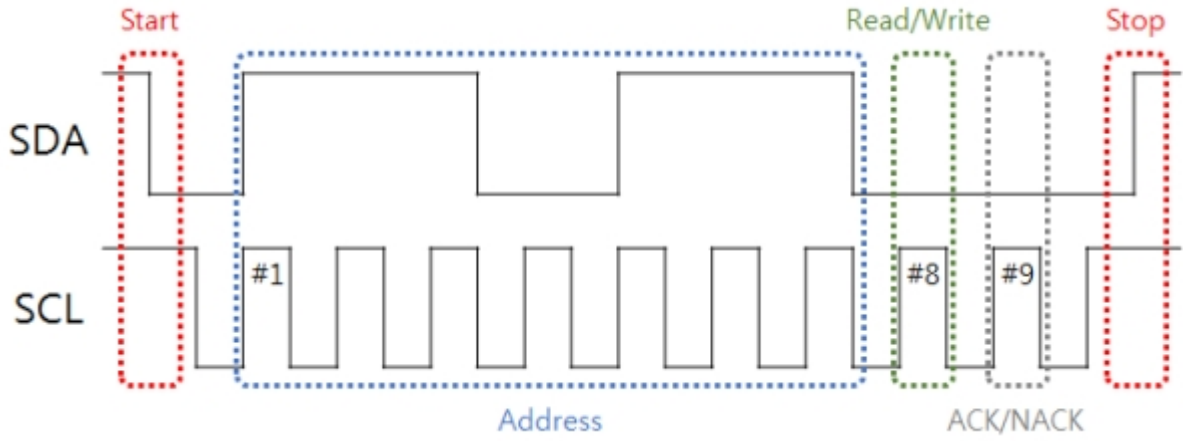
```
$wbuf = 0x00;                       // Data to be sent
$rbuf = "";

$pid = pid_open("/mmap/spi0");      // open SPI0
pid_ioctl($pid, "set mode 3");      // set SPI mode to 3
pid_ioctl($pid, "set lsb 0");       // set bit transmission order: MSB first
pid_write($pid, $wbuf, 1);          // write 1 byte to buffer: 0x00
pid_ioctl($pid, "req start");       // request to write data
while(pid_ioctl($pid, "get txlen")) // check the size of transmitted data
    ;
pid_read($pid, $rbuf, 1);           // read 1 byte
pid_close($pid);
```

I2C Overview

PHPoC provides I2C, two lines bus interface.

I2C Data Structure

I2C data is always sent in 8-bit unit. Structure of I2C data is as follows:



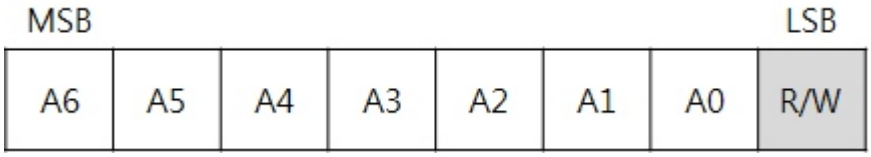
Start and Stop Conditions

Start and stop conditions of I2C are as follows:

Condition	SCL	SDA
Start	HIGH	HIGH > LOW
Stop	HIGH	LOW > HIGH

Slave Addressing

PHPoC uses 7-bit slave addressing. The LSB of address byte defines that the frame is for reading or for writing.



Read / Write Conditions

I2C communication consist of reading or writing data from a master.

Division	SCL	SDA
Read	8th bit - HIGH	HIGH
Write	8th bit - HIGH	LOW

ACK / NACK

Both an I2C master and a slave send acknowledgement (ACK) when receive 8 bits data. Correct acknowledgement is implemented by output HIGH on 9th bit. If the state of bus is HIGH, has not

received data yet.

Division	SCL	SDA
Acknowledgement(ACK)	9th bit - HIGH	LOW
No Acknowledgement (NACK)	9th bit - HIGH	HIGH

I2C Commuication Scenario

4 different scenarios of I2C communication are as follows. In the scenarios below, white background areas are a master's output and gray background areas are a slave's output.

Success to Write Data

Start	Device Address	R/W	ACK	Send Data	ACK	Stop
	1 1 1 0 1 1 1	0	0	0 0 0 1 1 1 1 0	0	

Fail to Write Data

Start	Device Address	R/W	ACK	Stop
	1 1 1 0 1 1 1	0	1	

Success to Read Data

Start	Device Address	R/W	ACK	Send Data	ACK	Stop
	1 1 1 0 1 1 1	1	0	0 1 1 0 0 1 1 0	0	

Fail to Read Data

Start	Device Address	R/W	ACK	Stop
	1 1 1 0 1 1 1	1	1	

Steps of Using I2C

General steps of using I2C is as follows:

Opening I2C

To open I2C, pid_open function is required.

```
$pid = pid_open("/mmap/i2c0");       // open I2C
```

※ Refer to Appendix for detailed I2C information.

Setting and Using I2C

To use or set I2C, pid_ioctl function is required. Available commands are as follows:

Command	Sub Command		Description
set	mode	sm	set data rate: standard - 100 Kbps
		fm	set data rate: fast - 400 Kbps
	daddr	[A]	set address: local device address
	saddr	[A]	set address: slave device address
get	rxlen		get the number of pending bytes in receive buffer
	txlen		get the number of pending bytes in send buffer
	error	nack	get the number of NACK in the last transaction
		bus	get the number of bus error in the last transaction
	state		0 - idle, etc. - I2C internal operation
req	read	[N]	request to read
	write	-	request to write
		wait	request to write and wait ("req stop" is required)
	stop		request to stop writing data ("req wait" is required)
	reset		request to reset bus

Setting I2C

You can set a communication mode and a device address by using a "set" command.

Set Data Rate

PHPoC provides standard mode and fast mode. The standard mode is the default value.

Communication Mode	Syntax
standard	pid_ioctl($pid, "set mode sm");
fast	pid_ioctl($pid, "set mode fm");

Set Device Address

An I2C master uses a device address to select a slave to send data and a local device address to set its own device address. How to set those addresses is as follows:

Type	Syntax
slave device address	pid_ioctl($pid, "set saddr [A]");
local device address	pid_ioctl($pid, "set daddr [A]");

Each device address should be entered in 2 digits hexadecimal number. LSB of this byte is always zero because PHPoC supports 7 bits addressing for I2C. Note that there are some reserved addresses. If you use those addresses, PHPoC will stop due to errors.

Addresses (Binary)								Example(Hexa)	Comment
7	6	5	4	3	2	1	0		
X	X	X	X	X	X	X	1	E1, A3, 1B	LSB is 1
0	0	0	0	X	X	X	X	00 ~ 0F	All of 4 Upper bits are 0
1	1	1	1	X	X	X	X	F0 ~ FF	All of 4 Upper bits are 1

Getting I2C Status

You can get status of I2C by using "get" command.

Get the Number of Pending Bytes in Send and Receive Buffer

How to get the number of pending bytes in send and receive buffer is as follows:

Division	Syntax
Send buffer	pid_ioctl($pid, "get txlen");
Receive buffer	pid_ioctl($pid, "get rxlen");

Get Amount of Errors

You can check the amount of bus errors and NACK in the last transaction.

Division	Syntax
NACK	pid_ioctl($pid, "get error nack");
bus Error	pid_ioctl($pid, "get error bus");

Get Status

How to get an I2C state is as follows: 0 will be returned in idle state and the other values will be returned if otherwise.

Division	Syntax
State	pid_ioctl($pid, "get state");

Using I2C

Request to Read Data

An I2C master requests to read data from an I2C slave with this command. After sending this data, you can read it by the slave by pid_read function.

Division	Syntax
Request to Read Data	pid_ioctl($pid, "req read [N]");

Request to Write Data

An I2C master requests to write data to an I2C slave with this command. There are two ways and those are as follows:

Division	Syntax
Request to Write Data	pid_ioctl($pid, "req write");
Request to Write Data and Wait	pid_ioctl($pid, "req write wait");
Stop Writing Data	pid_ioctl($pid, "req stop");

When "req write" command runs, a master immediately sends data. Thus, you have to input data to a buffer before running this command. On the other hand, "req wait" command changes bus state to start condition and does not change it to stop condition until "req stop" command runs. Thus, you can write data to a slave by using pid_write function before "req stop" command runs.

Request to Reset Bus

You can reset an I2C bus with this command when communication is bad.

Command	Syntax
Request to Request Bus	pid_ioctl($pid, "req reset");

Example of Using I2C

Writng Data to a Slave

The following code is a common example that an I2C master writes data to a slave.

example of writing data

```
$wbuf = 0x7A;
$pid = pid_open("/mmap/i2c0");       // open I2C 0
pid_ioctl($pid, "set mode fm");      // set data rate: fast mode
pid_ioctl($pid, "set saddr ee");     // set slave device address: 0xEE
pid_write($pid, $wbuf, 1);           // input a byte to buffer: 0x7A
pid_ioctl($pid, "req write");        // request to write
while(pid_ioctl($pid, "get txlen")) // check received data
    ;
pid_close($pid);
```

example of standing by and running of writing data

```
$pid = pid_open("/mmap/i2c0");        // open I2C 0
pid_ioctl($pid, "set mode fm");       // set data rate: fast mode
pid_ioctl($pid, "set saddr ee");      // set slave device address: 0xEE
pid_ioctl($pid, "req write wait");    // request to write and wait
pid_write($pid, 0x7A, 1);             // write 1 byte: 0x7A
pid_write($pid, 0x8A, 1);             // write 1 byte: 0x8A
pid_write($pid, 0x9A, 1);             // write 1 byte: 0x9A
while(pid_ioctl($pid, "get txlen"))  // check received data
    ;
pid_ioctl($pid, "req stop");          // stop writing data
pid_close($pid);
```

Reading Data from a Slave

The following code is a common example that an I2C master receives data from a slave.

example of reading data

```
$rbuf = "";
$pid = pid_open("/mmap/i2c0");          // open I2C
pid_ioctl($pid, "set mode fm");         // set data rate: fast mode
pid_ioctl($pid, "set saddr ee");        // set slave device address: 0xEE
pid_ioctl($pid, "req read 2");          // request to read 2 bytes
while(pid_ioctl($pid, "get rxlen") < 2) // check received data
    ;
```

```
pid_read($pid, $rbuf);                  // read buffer
pid_close($pid);
```

RTC Overview

PHPoC provides Real Time Clock (RTC) for keeping accurate time. RTC is powered by an internal battery show it keep running even when system power is off.

Steps of Using RTC

General steps of using RTC are as follows:

Opening RTC

To open RTC, pid_open function is required.

```
$pid = pid_open("/mmap/rtc0");        // open RTC
```

※ Refer to Appendix for detailed RTC information depending on the types of products.

Setting RTC Time

To set current time of RTC, "set date" command of pid_ioctl function is used.

```
pid_ioctl($pid, "set date TIME");
```

TIME is a string and the structure is as follows:

Division	Year	Month	Day	Hour	Minute	Second
Format	YYYY	MM	DD	hh	mm	ss
example 1	2000	01	03	03	05	07
example 2	2010	12	28	19	59	16

The following example shows how to set time.

example of setting RTC

```
$pid = pid_open("/mmap/rtc0");       // open RTC0
$date = "20160720135607";            // 13:56:07, 20th July, 2016
pid_ioctl($pid, "set date $date");  // set RTC time
pid_close($pid);
```

Getting RTC Time

To read current time from RTC, pid_ioctl function is used.

```
pid_ioctl($pid, "get ITEM");
```

Available RTC Information

ITEM	Description	Return Value	Return Type
date	date and time	e.g. 20160720135607	string
wday	day of week	0: Sun, 1: Mon, 2: Tue, 3: Wed, 4: Thu, 5: Fri, 6: Sat	integer

example of getting RTC information

```
$date = "";
$wday = 0;
$pid = pid_open("/mmap/rtc0");         // open RTC 0
$date = pid_ioctl($pid, "get date");  // get the date and time
$wday = pid_ioctl($pid, "get wday");  // get the day of week
pid_close($pid);
```

※ Return value of RTC time has the same structure with a value for setting time.

"date()" Function

PHPoC provides an internal function for getting RTC information. You can get RTC information with various formats by using this function.

example of getting RTC information by "date()" function

```
$date1 = date("Y-m-d H:i:s");
$date2 = date("D M j H:i:s Y");
echo "$date1₩r₩n";              // output e.g. 2016-07-20 13:56:07
echo "$date2₩r₩n";              // output e.g. Wed Jul 20 13:56:07 2016
```

※ Refer to the PHPoC Internal Functions document for detailed information of date function.

Device Related Functions

PHPoC provides a bunch of internal functions for using devices as follows:

Function	Use Format
pid_bind	pid_bind(PID[, IP, PORT]);
pid_close	pid_close(PID);
pid_connect	pid_connect(PID, IP, PORT);
pid_ioctl	pid_ioctl(PID, COMMAND);
pid_listen	pid_listen(PID, [BACKLOG]);
pid_open	pid_open(PID[, FLAG]);
pid_read	pid_read(PID, BUF[, LEN]);
pid_recv	pid_recv(PID, BUF[, LEN, FLAG]);
pid_recvfrom	pid_recvfrom(PID, BUF[, LEN, FLAG, IP, PORT]);
pid_send	pid_send(PID, BUF[, LEN, FLAG]);
pid_sendto	pid_sendto(PID, BUF[, LEN, FLAG, IP, PORT]);
pid_write	pid_write(PID, BUF[, LEN]);

※ Refer to the PHPoC Internal Functions document for detailed information of internal functions.

Device Information

The Number of Device Depending on Product Types

Device	P4S-341 / P4S-342
UART	2
NET	1
TCP	5
UDP	5
UIO	1 (24 CH)
ST	8
HT	4
ADC	2 (6 CH)
RTC	1
SPI	1
I2C	1

Device File Path Depending on Product Types

UART

Product	Path of the Device
P4S-341 / P4S-342	/mmap/uart0
	/mmap/uart1

NET

Product	Path of the Device	Note
P4S-341	/mmap/net0	Ethernet
P4S-342	/mmap/net1	Wireless LAN

TCP

Product	Path of the Device
P4S-341 / P4S-342	/mmap/tcp0
	/mmap/tcp1
	/mmap/tcp2
	/mmap/tcp3
	/mmap/tcp4

UDP

Product	Path of the Device
P4S-341 / P4S-342	/mmap/udp0
	/mmap/udp1
	/mmap/udp2
	/mmap/udp3
	/mmap/udp4

I/O

<table>
<tr><th>Product</th><th>Path of the Device and Mapping Information</th></tr>
<tr><td>P4S-341</td><td>/mmap/uio0
DIO LED Not Assigned
<table>
<tr><td>#7</td><td>#6</td><td>#5</td><td>#4</td><td>#3</td><td>#2</td><td>#1</td><td>#0</td></tr>
<tr><td>#15</td><td>#14</td><td>#13</td><td>#12</td><td>#11</td><td>#10</td><td>#9</td><td>#8</td></tr>
<tr><td>#23</td><td>#22</td><td>#21</td><td>#20</td><td>#19</td><td>#18</td><td>#17</td><td>#16</td></tr>
<tr><td>#31</td><td>#30</td><td>#29</td><td>#28</td><td>#27</td><td>#26</td><td>#25</td><td>#24</td></tr>
</table>
"/mmap/uio0"</td></tr>
<tr><td>P4S-342</td><td>/mmap/uio0
DIO LED Not Assigned
<table>
<tr><td>#7</td><td>#6</td><td>#5</td><td>#4</td><td>#3</td><td>#2</td><td>#1</td><td>#0</td></tr>
<tr><td>#15</td><td>#14</td><td>#13</td><td>#12</td><td>#11</td><td>#10</td><td>#9</td><td>#8</td></tr>
<tr><td>#23</td><td>#22</td><td>#21</td><td>#20</td><td>#19</td><td>#18</td><td>#17</td><td>#16</td></tr>
<tr><td>#31</td><td>#30</td><td>#29</td><td>#28</td><td>#27</td><td>#26</td><td>#25</td><td>#24</td></tr>
</table>
"/mmap/uio0"</td></tr>
</table>

ST

Product	Path of the Device
P4S-341 / P4S-342	/mmap/st0
	/mmap/st1
	/mmap/st2
	/mmap/st3
	/mmap/st4
	/mmap/st5
	/mmap/st6
	/mmap/st7

HT

Product	Path of the Device
P4S-341 / P4S-342	/mmap/ht0
	/mmap/ht1
	/mmap/ht2
	/mmap/ht3

ADC

Product	Path of the Device
P4S-341 / P4S-342	/mmap/adc0
	/mmap/adc1

SPI

Product	Path of the Device
P4S-341 / P4S-342	/mmap/spi0

I2C

Product	Path of the Device
P4S-341 / P4S-342	/mmap/i2c0

RTC

Product	Path of the Device
P4S-341 / P4S-342	/mmap/rtc0

ENV and User Memory

Product		Path of the Device	Size (Byte)
P4S-341 / P4S-342	System ENV	/mmap/envs	1536
	User ENV	/mmap/envu	1536
	User Memory	/mmap/um0	64
		/mmap/um1	64
		/mmap/um2	64
		/mmap/um3	64

F/W Specification and Restriction

Firmware

Firmware	Product
P40	P4S-341, P4S-342

Specification

Item	Value	Description
ENVS	1,536	Size of System ENV, byte
ENVU	1,536	Size of User ENV, byte
WLAN	1	Wireless LAN
EMAC	1	Ethernet
UART	2	The number of UART
FLOAT	Support	Floating Point Numbers
SSL	Support	SSL communication
PHP_MAX_NAME_SPACE	16	The number of Namespace
PHP_NAME_LEN	32	Size of User Identifier
PHP_MAX_USER_DEF_NAME	480	The number of User Identifier
PHP_LLSTR_BLK_SIZE	64	Size of String Block, byte
PHP_MAX_LLSTR_BLK	192	The number of String Blocks
string buffer size	12K	Size of string buffer, byte
PHP_MAX_STRING_LEN	1,536	Size of string variable, byte
PHP_INT_MAX	≒ $9.2*10^{18}$	Max value of integer type
EZFS_MAX_NAME_LEN	64	Size of EZFS filename, byte
TASK	2	The number of Task
TCP	5	The number of TCP
UDP	5	The number of UDP
TCP_RXBUF_SIZE	1,068	TCP receive buffer size
TCP_TXBUF_SIZE	1,152	TCP send buffer size
PDB_TXBUF_SIZE	2,048	PHPoCD send buffer size
HTTP_TXBUF_SIZE	1,536	HTTP send buffer size
UART_RXBUF_SIZE	1,024	UART send/receive buffer size
UDP_RXBUF_SIZE	512	UDP receive buffer size
ST	8	Software Timer
HT	4	Hardware Timer
ADC	2	Analog Input (ADC)
SPI	1	SPI
I2C	1	I2C
RTC	1	RTC

Limitations

Item	limitation
Level of Namespace	PHP_MAX_NAME_SPACE - 1
Level of Function Call	PHP_MAX_NAME_SPACE - 2
Size of User Identifier	PHP_NAME_LEN - 1
Size of String Variable	PHP_MAX_STRING_LEN - 2
Size of Array Offset	string length - 2
Size of Filename	EZFS_MAX_NAME_LEN - 1
Size of arguments for system function	PHP_LLSTR_BLK_SIZE - 1
Size of arguments for pid_ioctl function	PHP_LLSTR_BLK_SIZE - 1
Size of $address of function sendto	PHP_LLSTR_BLK_SIZE - 1
Size of $needle & $replace of function str_replace	PHP_LLSTR_BLK_SIZE - 1

Item	limitation
Size of $address of function inet_pton	PHP_LLSTR_BLK_SIZE - 1
Size of $address of function inet_ntop	PHP_LLSTR_BLK_SIZE - 1
Size of $delimiter of function explode	PHP_LLSTR_BLK_SIZE - 1
Maximum size of UDP data for receiving	UDP receive buffer size - 2