

# P7 library

## Contents

Introduction .....	3
Directory structure .....	4
Components overview .....	5
Trace .....	6
Telemetry .....	7
Speed tests .....	9
Initialization parameters .....	11
P7 library interfaces .....	13
Client interface .....	13
C++ interface .....	14
C interface .....	16
C# interface .....	18
Python interface .....	20
Trace interface .....	21
Trace verbosity levels .....	21
Trace format string .....	21
C++ interface .....	22
C interface .....	26
C# interface .....	32
Python interface .....	35
Telemetry interface .....	38
Configuration .....	38
C++ interface .....	38
C interface .....	41
C# interface .....	44
Python interface .....	46
P7 instances redistributing .....	49
Process crush handling .....	50
Compilation .....	51
Shared library (dll, so) export functions .....	52
Examples .....	53

## Introduction

P7 is cross platform library for handling your needs in delivering & storing your trace/log messages and telemetry data (CPU, memory, buffers utilization, threads cyclograms, etc.)

Basic facts:

- C++/C/C#/Python interfaces are available
- Cross platform (Linux x86/x64, Windows x86/x64)
- Speed is priority, library designed to suit high load (for details see [speed measurement](#) chapter), for example average performance for Intel i7-870 is:
  - 50 000 traces per second - 0,5% CPU, max ~3.3 million traces per second to network, ~5.3 million traces per second to file
  - 110 000 samples per second - 0,5% CPU, max ~3.8 million per second to network, ~6.0 million per second to file
- Thread safe
- Unicode support (UTF-8 for Linux, UTF-16 for Windows)
- No external dependencies
- High-resolution time stamps (resolution depends on HW high-resolution performance counter, usually it is 100ns)
- Different sinks (transport & storages) are supported:
  - Network (Baical server)
  - File
  - Auto (Baical server if reachable, else – file)
  - Null
- Files rotation setting (by size or time)
- Files max count setting (deleting old files automatically)
- Remote management from Baical server (set verbosity per module, enable/disable telemetry counters)
- Providing maximum information for every trace message:
  - Format string
  - Function name
  - File name
  - File line number
  - Module ID & name (if it was registered)
  - Trace ID
  - Sequence Number
  - Variable arguments
  - Level (error, warning, .. etc.)
  - Time with 100ns granularity
  - Current thread ID
  - Current thread name (if it was registered)
  - Current processor number
- Shared memory is used – create your trace and telemetry channels once and access it from any process module or class without passing handles
- Simple way (one function) to flush all P7 buffers for all P7 objects in case of process crush
- Trace & telemetry files have binary format (due to speed requirements – binary files much more compact than raw text), export to text is available

## Directory structure

- Examples (folder with examples for different languages)
  - C
  - C#
  - Cpp
  - Python
- Documentation (folder with library documentation)
- Headers (interfaces headers folder, use it for library integration)
  - GTypes.h – main types which are used by P7 (C/C++)
  - P7\_Client.h – P7 client interface (C++ only)
  - P7\_Telemetry.h – P7 telemetry interface (C++ only)
  - P7\_Trace.h – P7 trace interface (C++ only)
  - P7\_Extensions.h – P7 extensions types (C++ only)
  - P7\_Cproxy.h - P7 client, trace & telemetry proxy interface for C language, use it with static library (lib/a) or shared library (dll/so) integration
- Shared (used for library compilation, not necessary for library integration)
- Sources (source code of the library)
- Tests (library tests)
  - Speed – check the speed of traces & telemetry delivering on your hardware
  - Trace – few stability tests joined into one console application
- Wrappers (interfaces for foreign languages)
  - C#
  - Py
- build.sh – Linux compilation script, builds library itself (static & shared), examples, tests
- License.txt – library license
- P7.sln – Visual Studio 2010 solution

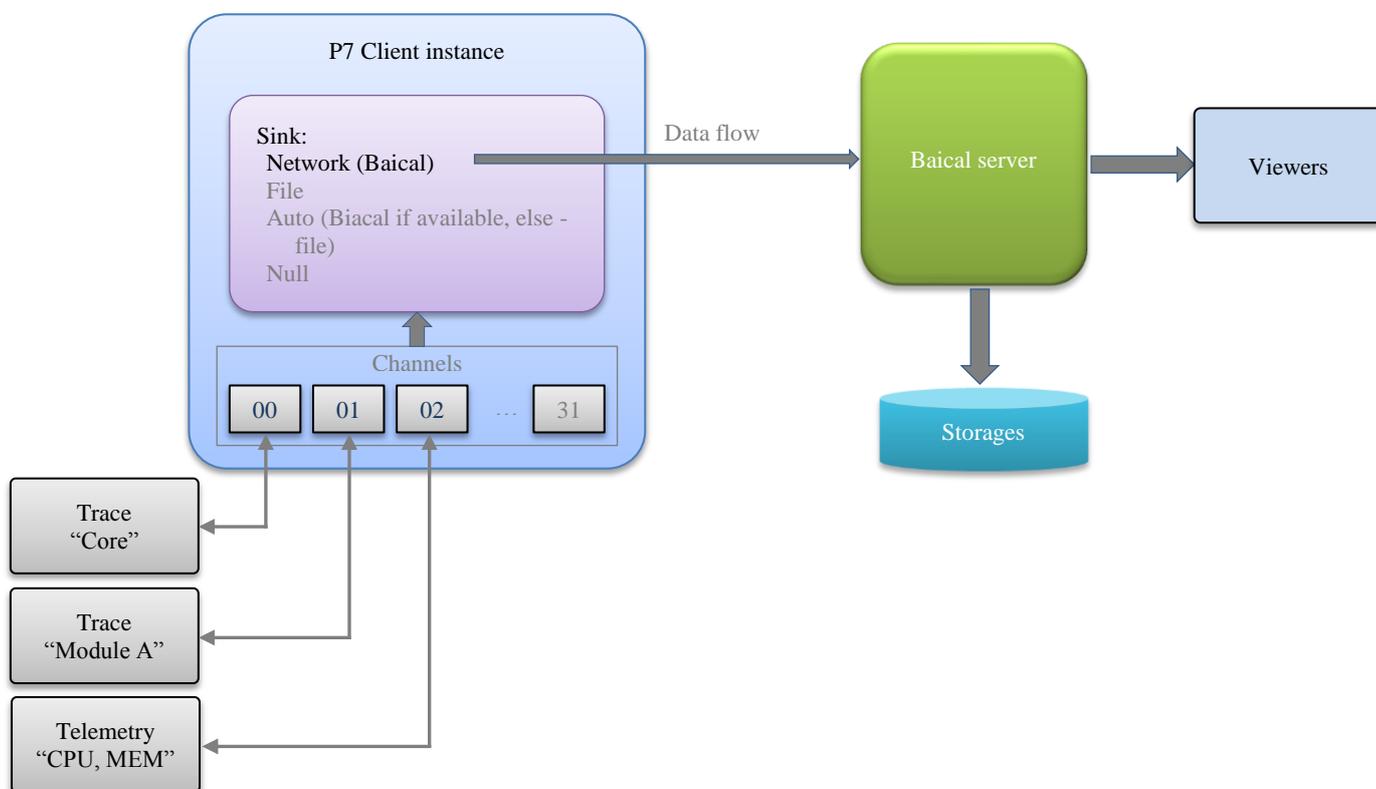
## Components overview

Internally P7 has very simple design, and consist of few modules:

- Channel – named data channel, used for wrapping user data into internal P7 format. For now there are next channels types are available:
  - Telemetry
  - Trace
- Sink – module which provides a way of delivering serialized data from channels to final destination. Next types are available for now:
  - Network – deliver data directly to Baical server
  - File – writes all user data into single binary file
  - Auto – delivers to Baical server if it is reachable otherwise to file
  - Null – drops all incoming data, save CPU for the hosting process
- Client – is a core module, it aggregate sink & channels together and manage them. Every client object can handle up to 32 independent channels

Let's take an example (diagram below) – developed application has to writes 2 independent log (trace) streams and 1 telemetry stream, and delivers them directly to Baical. Initialization sequence will be:

1. First of all you need to create P7 Client, and specify parameters for sink `"/P7.Sink=Baical /P7.Addr=127.0.0.1"`
2. Using the client create:
  - a. create first trace channel with name "Core"
  - b. create second trace channel named "Module A"
  - c. create telemetry channel named "CPU, MEM"



## Trace

From software engineer point of view trace is a source code line (function with variable arguments list):

```
IP7_Client *_l_iClient = P7_Create_Client(TM("/P7.Sink=Baical /P7.Addr=127.0.0.1"));
IP7_Trace *_l_iTrace = P7_Create_Trace(_l_iClient, TM("TraceChannel"));

_l_iTrace->P7_TRACE(0, TM("Test trace message %#d"), 0);
...
```

And at another side it looks like that (internal Baical logs):

#	ID	Level	Module	Proc.	Thread	Time	Text
17	1026	Info	0	1	Main (0x1758)	56:25.351'091"4	Use address = [::1]:9009
18	1026	Info	0	1	Main (0x1758)	56:25.351'377"1	Use address = 127.0.0.1:9009
19	1027	Info	Srv/Addr (6)	0	SrvAddr (0x060c)	56:25.351'433"1	[8] Start ...
20	1027	Info	Srv/Addr (6)	0	SrvAddr (0x0d60)	56:25.351'696"6	[9] Start ...
23	1028	Info	Baical (1)	0	Core (0x0d14)	56:25.355'696"3	Start core baical thread
24	1029	Info	Srv/Addr (6)	1	SrvAddr (0x0d60)	56:25.874'559"7	[9] New client, ID = 65535, Prev. connection addr = 0
26	1031	Info	Srv/Addr (6)	1	SrvAddr (0x0d60)	56:25.874'559"9	[9] Add connection 9000001, Client ID = 0, List lengt...
28	1033	Info	Prv/Net (3)	0	Core (0x0d14)	56:25.881'270"3	[0x019569B0] Create connection, 0x03833018, number = 1
29	1034	Info	Prv/Net (3)	0	Core (0x0d14)	56:25.891'476"8	[0x019569B0] Create channel, ID 1, Channel 0
30	1035	Info	Baical (1)	0	Core (0x0d14)	56:25.891'482"7	Add stream ID = 1
31	1034	Info	Prv/Net (3)	0	Core (0x0d14)	56:25.893'570"5	[0x019569B0] Create channel, ID 2, Channel 1
32	1035	Info	Baical (1)	0	Core (0x0d14)	56:25.893'574"8	Add stream ID = 2
3954	1038	Warning	U1 (14)	1	U1 (0x1614)	01:30.541'695"6	Unable to establish connection, try again
4708	1039	Trace	Tel/vwr/Rnd (12)	1	Main (0x1758)	19:25.731'736"9	[0x068929E0] Render cmd, offset = 0

P7P.cpp  
CP7Shell::Create\_Channel, line 1059

It is very similar to logging, but unlike logging - trace gives you much more freedom, you don't have to choose which information to write, you may write everything (without impacting on application performance, 50k traces per second with 0.5% CPU for example, for details see [Speed test](#) chapter) and then during debugging session use flexible filtering engine to find interesting parts, in this case you will be sure that all necessary information is available for you.

This approach became possible due to P7 performance. Trace module was designed with the idea of performance, especially on small embedded system.

To be able to send so much information next optimizations are used:

- Do not deliver & records duplicated information every time – the most heavy text fields [Format string, Function name, File name, File line number, Module ID] are delivered & recorded once - only for first call (the same information will be transmitted once in case of new connection establishing)
- Do not format trace string on client side, variable arguments formatting is a heavy operation and it will be done on server side by request
- Deliver only changes for every subsequent trace call [variable arguments, sequence number, time with 100ns granularity, current thread, processor core number]

**N.B.:** The **best performance** is provided by C++ and C interfaces ([release build](#)), C# & Python wrappers provides less performing solutions.

## Telemetry

From software engineer's point of view telemetry is a few source code lines:

```
IP7_Client      *l_hClient      = P7_Create_Client(TM("/P7.Sink=Baical /P7.Addr=127.0.0.1"));
IP7_Telemetry  *l_hTelemetry  = P7_Create_Telemetry(l_hClient, TM("AppStatistics"));
tUINT8         l_bCpuId       = 0;
tUINT8         l_bMemId       = 0;
tINT64         l_llCPU        = 0;
tINT64         l_llMem        = 0;

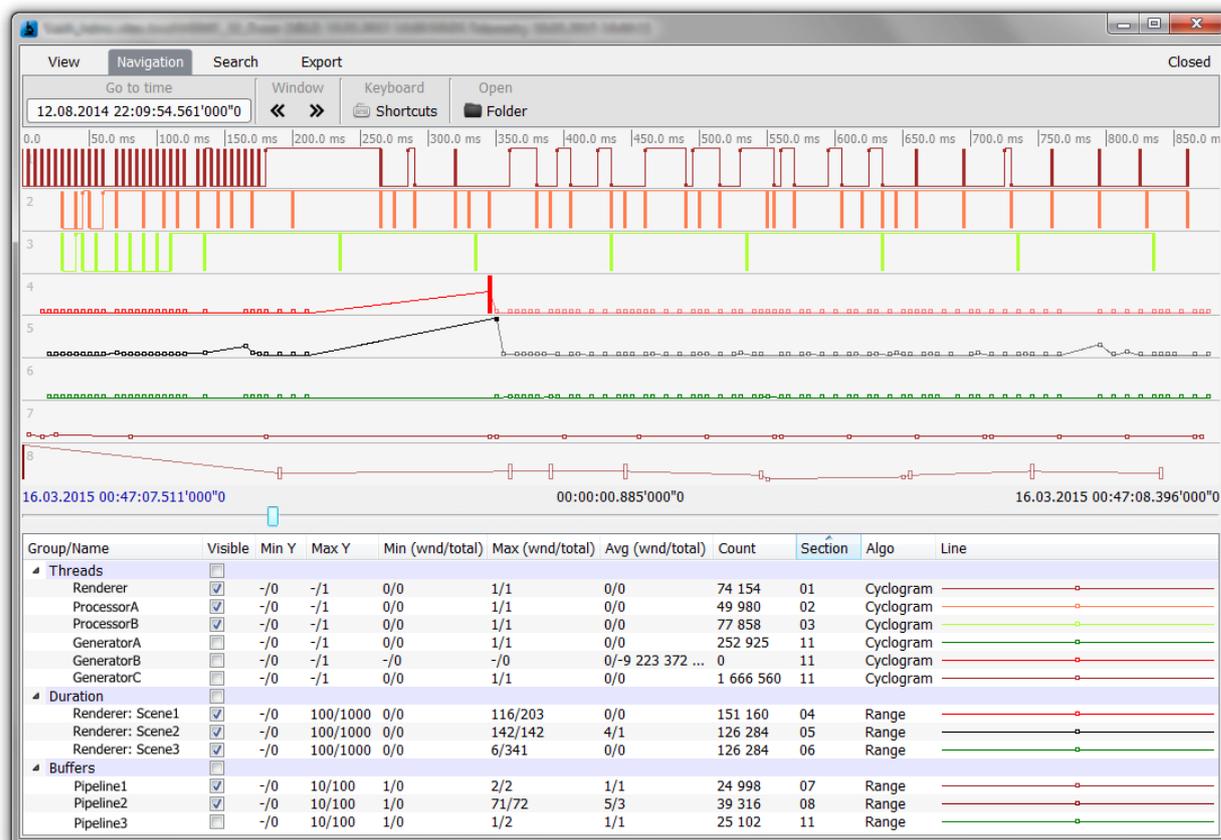
l_hTelemetry->Create(TM("System/CPU"), 0, 100, 90, 1, &l_bCpuId);
l_hTelemetry->Create(TM("System/Mem(mb)"), 0, 500, 450, 1, &l_bMemId);

while (/* ... */)
{
    //query in cycle current CPU & mem values ...
    //l_llCPU = Get_CPU_Utilization()
    //l_llMem = Get_Mem_Utilization()

    //deliver info
    l_pTelemetry->Add(l_bCpuId, l_llCPU);
    l_pTelemetry->Add(l_bMemId, l_llMem);

    //do something ...
}
```

And at another side it looks like that:



Telemetry is a simple and fast way to record any dynamically changed values for further or real time analysis on Baical server side. You may use it for a lot of things: system statistics (cpu, memory, hdd, etc.), buffers filling, threads cyclograms or synchronization, mutexes, networks delays, packets sizes, etc. There are plenty of possible usage cases.

Some facts about telemetry:

- Every telemetry channel can handle up to 256 independent counters

- No (or minimal) impact on application performance – on modern hardware (2014) spend only 300 ns for processing one telemetry sample (add(...) -> network -> Baical srv -> HDD), it is about 220 000 of samples per second with about 1% CPU usage
- You can enable or disable counters online from Baical server – it allows you visualize and record only necessary data
- Every telemetry sample contains 64 bit signed value & high resolution time stamp

**N.B.:** The **best performance** is provided by C++ and C interfaces, C# & Python wrappers provides less performing solutions.

## Speed tests

P7 library was designed with the idea of performance, such approach allows software engineer to deliver maximum information about program execution in real-time with minimum resources consumption, and next few tests on different platforms will have to confirm this statement.

Test conditions:

- Test application sent traces & telemetry data in cycle & make time measurement
- P7 library use next option `/P7.Sink=Baical`, this means all data goes through network interface to the Baical server (loopback network interface is used)
- Every trace messages contains next fields: format string, function name, file name, file line number, module ID, variable arguments, sequence number, time with 100ns granularity, current thread, module ID, processor core number
- Every telemetry sample contains next fields: counters ID, sample value, sample time with 100ns granularity
- Baical server will receive & save incoming data

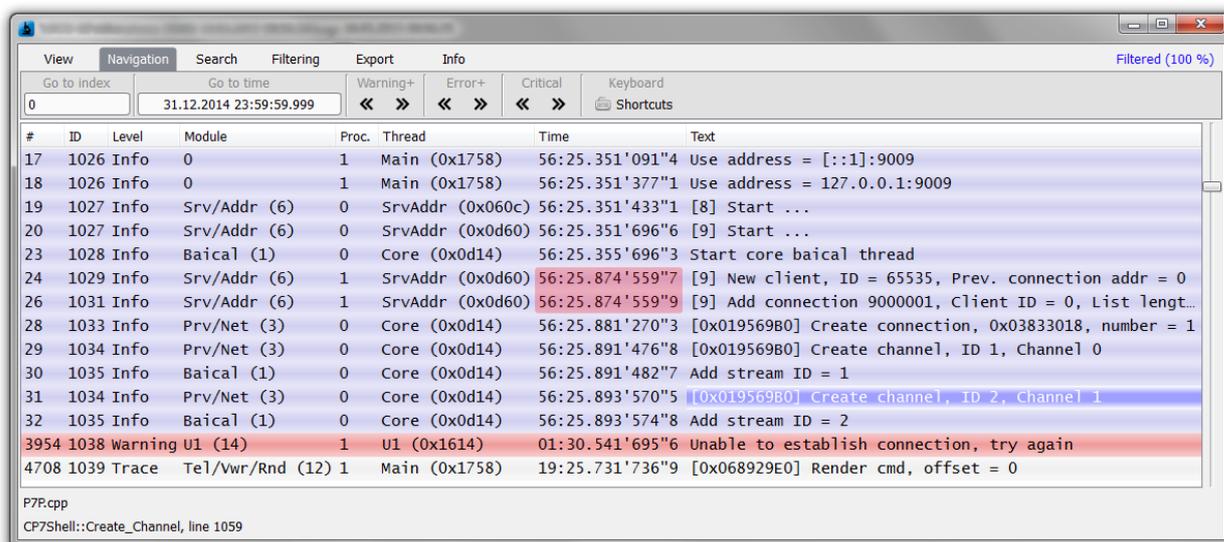
Test results for trace channel:

- *ARM 926EJ (v5)* - 1 000 per second - 0,5% CPU, ~20 000 per second max
- *Intel E8400 (Core 2 duo)* -15 000 per second - 0,5% CPU, ~750 000 per second max
- *Intel i7-870* - 50 000 per second - 0,5% CPU, ~3.3 million per second max

Test results for telemetry channel:

- *ARM 926EJ (v5)* - 2 000 samples per second - 0,5% CPU, ~50 000 per second max
- *Intel E8400 (Core 2 duo)* - 25 000 samples per second - 0,5% CPU, ~1.2 million per second max
- *Intel i7-870* - 110 000 samples per second - 0,5% CPU, ~3.8 million per second max

Next screenshot shows delay between 2 trace messages about 200 nanoseconds on modern hardware (2014):

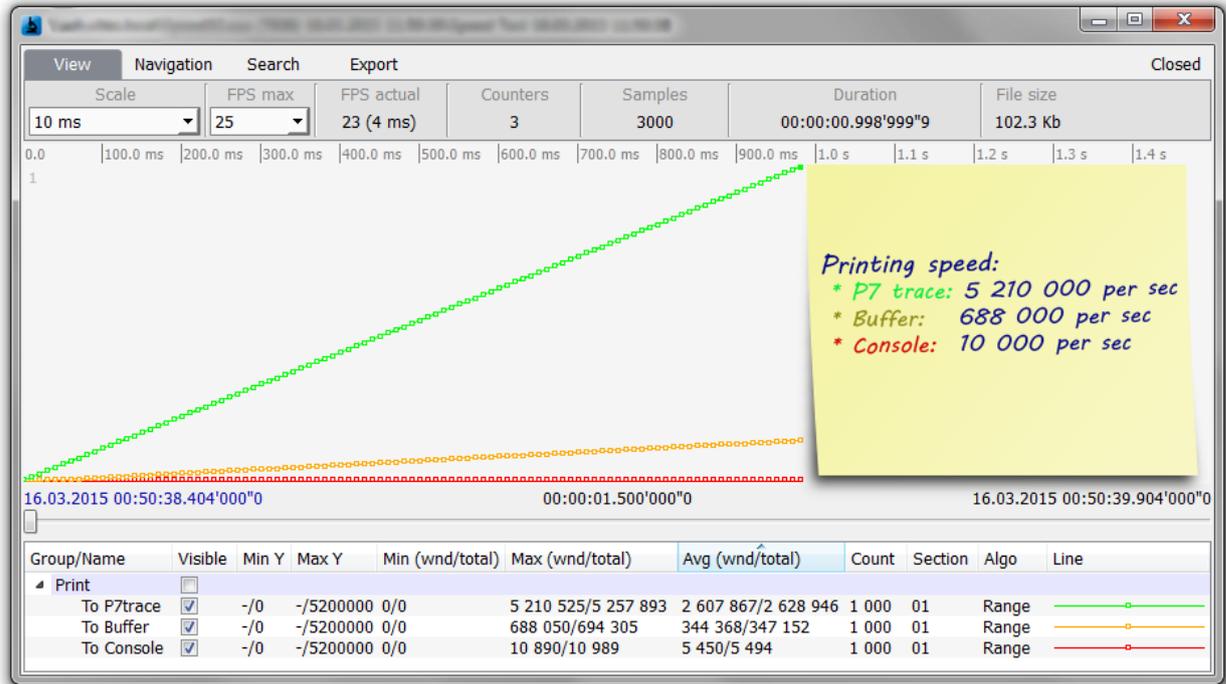


You may build & run your own speed tests to estimate library performance on your hardware and compare the performance of printing to the memory buffer, console and P7 trace channel, to do that you need:

- Compile under Linux or Windows project `<P7 folder>\Tests\Speed`. You can do it by using `P7.sln` for visual studio or linux shell script – `build.sh`
- Generated binaries are located in `<P7 folder>\Binaries`
- Run the generated binary (Windows: `Speed64.exe/Speed32.exe`, Linux: `Speed`)

- Record to local file: >SpeedXXX /P7.Sink=File
- Deliver to Baical server: >SpeedXXX /P7.Sink=Baical /P7.Addr=127.0.0.1

Next diagram shows the test result on Intel i7-870 platform and saving trace messages to file:



## Initialization parameters

Initialization parameters is a string like: `"/P7.Sink=Baical /P7.Addr=127.0.0.1 /P7.Pool=4096"`

Initialization parameters are used by every instance of P7 client - when you are going to create your P7 client instance you have to specify parameters for it or pass empty/NULL string to use default values.

You may pass hardcoded parameters directly to the client like that:

```

////////////////////////////////////
//C++
#include "GTypes.h"
#include "P7_Client.h"
int main(int i_iArgC, char* i_pArgV[])
{
    IP7_Client *l_pClient = P7_Create_Client(TM("/P7.Sink=Baical"));
    //...
}
////////////////////////////////////

//C
#include "GTypes.h"
#include "P7_Cproxy.h"
int main(int i_iArgC, char* i_pArgV[])
{
    hp7_Client *l_pClient = P7_Client_Create(TM("/P7.Sink=Baical"));
    //...
}
////////////////////////////////////

//C#
using P7;
namespace CSharp_Example
{
    class Program
    {
        static void Main(string[] args)
        {
            //initialize P7 client
            P7.Client l_pClient = new P7.Client("/P7.Sink=Baical");
        }
    }
}
////////////////////////////////////

#Python
import P7

P7.Register_Client(P7.UTF(u"MyClient"), P7.UTF(u"/P7.Sink=Baical"))
#////////////////////////////////////

```

Or you may pass parameters through command line (if you are using both modes – **console parameters have priority over hardcoded parameters**):

```
>> MyApplication.exe /P7.Sink=Baical /P7.Addr=localhost
```

Next parameters are common for all possible sink:

- `"/P7.Sink"` - Select data flow direction, there are few values:
  - `"Baical"` – deliver to Baical server over network
  - `"File"` – into a file
  - `"Auto"` – deliver to Baical if connection is established, otherwise to file (connection timeout is 250 ms)
  - `"Null"` - all data will be dropped

Default value is `"Baical"`. Example: `"/P7.Sink=Auto"`

- `"/P7.Name"` – P7 client instance name, max length is about 96 characters, by default name of host process is used (preferred mode). For script languages where host process is script interpreter you may use this option. Example: `"/P7.Name=MyChannel"`
- `"/P7.On"` – option allows enable/disable P7 client, by default P7 is on (1). Example `"/P7.On=0"`
- `"/P7.Verb"` – P7 library has internal logging mechanism(OFF by default), using this option you can set logging verbosity and automatically enable logging, next values are available:
  - "0" – info
  - "1" – debug
  - "2" – warning
  - "3" – error
  - "4" – critical

For example `"/P7.Verb=0"`. For Linux all P7 internal logs will be redirected to console stdout, for Windows folder `"P7.Logs"` will be created in host process folder and all further logs will be stored there.

- `"/P7.Pool"` – set maximum memory size available for internal buffers in kilobytes. Minimal 16(kb), maximal is limited by your OS and HW, default value is 4096 (4mb). Example if 1Mb allocation: `"/P7.Pool=1024"`
- `"/P7.Help"` – print console help

Next parameters are applicable for `"/P7.Sink=Baical"` or `"/P7.Sink=Auto"`:

- `"/P7.Addr"` – set Baical server network address (IPV4, IPV6, NetBios name). Example: `"/P7.Addr>::1"`, `"/P7.Addr=127.0.0.1"`, `"/P7.Addr=MyPC"`
- `"/P7.Port"` – set Baical server listening UDP port (default is 9010), example: `"/P7.Port=9010"`
- `"/P7.PSize"` – set transport packet size. Min value is 512 bytes, Max - 65535, Default – 512. Example: `"/P7.PSize=1472"`. Bigger packet allows transmit data with less overhead, but if you specify packet larger than your network MTU – there is a risk of transmission losses. P7 network protocol handles packets damaging and loss and retransmit necessary data chunks, but if packet is bigger than MTU – P7 can't correctly process such situation for now.
- `"/P7.Window"` – size of the transmission window in packets, used to optimize transmission speed, usually it is not necessary to modify this parameter. Min value – 1, max value – ((pool size / packet size) / 2).
- `"/P7.Eto"` –transmission timeout (in seconds) when P7 object has to be closed. Usage scenario:
  - Application sending data to Baical server through P7
  - For some reasons connection with Baical has been lost
  - Some data are still inside P7 buffers and P7 tries to deliver it
  - Application is closed by user and `"/P7.Eto"` value is used to specify time in second during which P7 will attempts to deliver data reminder.

Next parameters are applicable for `"/P7.Sink=File"` or `"/P7.Sink=Auto"`:

- `"/P7.Dir"` – option allows to specify directory where P7 files will be created, if it is not specified process directory will be used, examples: `"/P7.Dir=/home/user/logs/"`, `"/P7.Dir=C:\Logs\"`
- `"/P7.Roll"` – use option to specify files rolling value & type. There are 2 rolling types:
  - Rolling by file size, measured in megabytes ("mb" command postfix)
  - Rolling by time, measured in hours, 1000 hours max ("hr" command postfix)
 Examples: `"/P7.Roll=100mb"`, `"/P7.Roll=24hr"`
- `"/P7.Files"` – option defines maximum P7 logs files in destination folder, in case if count of files is larger than specified value - oldest files will be removed. Default value is OFF (0), max value – 4096. Example: `"/P7.Files=4096"`

## P7 library interfaces

### Client interface

---

Client is a core module of P7 library, working with the library start form client creation. Client is responsible for delivering your traces & telemetry data into final destination.

Every client object can handle up to 32 independent channels.

Number of clients per process is limited by available memory.

## C++ interface

Client header file is located in <P7>/Headers/P7\_Client.h

### P7\_Create\_Client

Function allows to create P7 client object

```
IP7_Client *P7_Create_Client(const tXCHAR *i_pArgs)
```

*Parameters:* argument string, see "[Initialization parameters](#)" chapter for details

*Return:*

- Valid pointer to [IP7\\_Client](#) interface in case of success
- NULL in case of failure

### P7\_Get\_Shared

This functions allows you to get P7 client instance if it was created by someone else inside current process and shared using [IP7\\_Client::Share\(...\)](#) function.

Sharing mechanism is very flexible way to redistribute your [IP7\\_Client](#) object among your modules without passing pointer to it and modification your interfaces, function is thread safe.

```
IP7_Client *P7_Get_Shared(const tXCHAR *i_pName)
```

*Parameters:* name of previously shared P7 client instance

*Return:*

- Valid pointer to [IP7\\_Client](#) interface in case of success
- NULL in case of failure

N.B.: Every successful call of this function increase reference counter value on retrieved [IP7\\_Client](#) object, **do not forget to call [Release\(\)](#) function**

### P7\_Exceptional\_Flush

Function allows to flush (deliver) not delivered/saved P7 buffers for all opened P7 clients and related channels owned by process in CASE OF your app/proc. crush. This function does not call system memory allocation functions only write to file/socket.

Classical scenario: your application has been crushed you catch the moment of crush and call this function once.

To read more about this function & usage scenario you may in chapter [Process crush handling](#)

**N.B.: DO NOT USE OTHER P7 FUNCTION AFTER CALLING THIS FUNCTION**

```
void P7_Exceptional_Flush()
```

### IP7\_Client::Add\_Ref

Function increase object reference counter

```
tINT32 Add_Ref()
```

*Return:* object's reference counter new value

### *IP7\_Client::Release*

Function decrease object reference counter, object will be destroyed when reference counter is equal to 0.

```
tINT32 Release()
```

*Return:* object's reference counter new value

### *IP7\_Client::Share*

Function allows to share current client instance in address space of current process. Sharing mechanism is very flexible way to redistribute your *IP7\_Client* object among your modules without passing pointer to it and modification your interfaces, function is thread safe.

```
tBOOL Share(const tXCHAR *i_pName)
```

*Parameters:* name of shared P7 client instance, should be unique

*Return:*

- TRUE – success
- FALSE – failure, the other object with the same name is already shared inside current process

## C interface

Client header file is located in <P7>/Headers/P7\_Cproxy.h

### P7\_Client\_Create

Function allows to create P7 client object

```
hp7_client P7_Client_Create(const tXCHAR *i_pArgs)
```

*Parameters:* argument string, see “[Initialization parameters](#)” chapter for details

*Return:*

- Valid handle of P7 client object in case of success
- NULL in case of failure

### P7\_Client\_Get\_Shared

This functions allows you to get P7 client instance if it was created by someone else inside current process and shared using [P7\\_Client\\_Share\(...\)](#) function.

Sharing mechanism is very flexible way to redistribute your P7 client object among your modules without passing pointer to it and modification your interfaces, function is thread safe.

```
hp7_client P7_Client_Get_Shared(const tXCHAR *i_pName)
```

*Parameters:* name of previously shared P7 client instance

*Return:*

- Valid handle of P7 client object in case of success
- NULL in case of failure

N.B.: Every successful call of this function increase reference counter value on retrieved [hp7\\_Client](#) object, **do not forget to call [P7\\_Client\\_Release\(...\)](#) function**

### P7\_Exceptional\_Flush

Function is described in C++ chapter [P7\\_Exceptional\\_Flush](#).

### P7\_Client\_Add\_Ref

Function increase object reference counter

```
tINT32 P7_Client_Add_Ref(hp7_client i_hClient)
```

*Parameters:* P7 client handle

*Return:* object’s reference counter new value

### P7\_Client\_Release

Function decrease object reference counter, object will be destroyed when reference counter is equal to 0.

```
tINT32 IP7_Client::Release(hp7_client i_hClient)
```

*Parameters:* P7 client handle

*Return:* object's reference counter new value

### ***P7\_Client\_Share***

Function allows to share current client instance in address space of current process. Sharing mechanism is very flexible way to redistribute your P7 client object among your modules without passing pointer to it and modification your interfaces, function is thread safe.

```
tBOOL P7_Client_Share(hP7_Client i_hClient, const tXCHAR *i_pName)
```

*Parameters:*

- *i\_hClient* – P7 client object handle
- *i\_pName* – name of shared P7 client instance, should be unique

*Return:*

- TRUE – success
- FALSE – failure, the other object with the same name is already shared inside current process

## C# interface

---

C# shell file is located in <P7>/ Wrappers/C#/P7.cs

C# shell depending on P7x64.dll/ P7x32.dll you may generate them by building P7 solution

### P7.Client

---

Constructor allows to create P7 client object

```
P7.Client(String i_sArgs)
```

*Parameters:* argument string, see "[Initialization parameters](#)" chapter for details

*Return:*

- Valid P7 client class instance in case of success
- ArgumentNullException(...) in case of failure

### P7.Get\_Shared

---

This functions allows you to get P7 client instance if it was created by someone else inside current process and shared using `P7::Client::Share(...)` function.

Sharing mechanism is very flexible way to redistribute your P7 client object among your modules without passing pointer to it and modification your interfaces, function is thread safe.

```
static P7.Client Get_Shared(String i_sName)
```

*Parameters:* name of previously shared P7 client instance

*Return:*

- Valid P7::Client instance in case of success
- null in case of failure

### P7.Exceptional\_Flush\_Buffers

---

Function allows to flush (deliver) not delivered/saved P7 buffers for all opened P7 clients and related channels owned by process in CASE OF your app/proc. crush.

Function is completely described in C++ chapter [P7 Exceptional Flush](#).

```
static void P7.Exceptional_Buffers_Flush()
```

### P7.Client.Add\_Ref

---

Function increase object reference counter

```
System.Int32 AddRef()
```

*Return:* object's reference counter new value

### P7.Client.Release

---

Function decrease object reference counter, object will be destroyed when reference counter is equal to 0.

```
System.Int32 Release()
```

*Return:* object's reference counter new value

### *P7.Client.Share*

Function allows to share current client instance in address space of current process. Sharing mechanism is very flexible way to redistribute your P7 client object among your modules without passing pointer to it and modification your interfaces, function is thread safe.

```
bool Share(String i_sName)
```

*Parameters:* name of shared P7 client instance, should be unique

*Return:*

- true – success
- false – failure, the other object with the same name is already shared inside current process

## Python interface

Python shell file is located in <P7>/ Wrappers/Py/P7.py

Python shell depending on P7x64.dll/P7x32.dll (Windows) and libP7.so (Linux) you may generate them by building P7 solution under Windows or run build.sh under Linux

## Importing

To import P7 Python shell you have to specify “P7\_Bin” environment variable, using that path P7 python shell will find P7 dll/so shared library, and update system path by P7.py directory.

Here is an example:

```
import sys;
import os;

#-----
#Loading P7 module by relative path and update PATH system env. variable to be
#able to load P7Client_XX.dll/so
l_sPath = os.path.dirname(os.path.realpath(__file__)) + "../../../Wrappers/Py/";
if l_sPath not in sys.path:
    sys.path.insert(0, l_sPath)

l_sPath = os.path.dirname(os.path.realpath(__file__)) + "../../../Binaries/";
if l_sPath not in os.environ['PATH']:
    os.environ['PATH'] += os.pathsep + l_sPath;
os.environ['P7_BIN'] = l_sPath;

import P7;
```

## P7.Register\_Client

Function allows to register P7 client in address space of current python session (execution one or group of depended scripts)

```
bool P7.Register_Client(i_sName, i_sArgs = None)
```

Parameters:

- i\_sName – name of P7 client, should be unique, used for sharing P7 client in address space of current python session
- i\_sArguments – argument string, see “[Initialization parameters](#)” chapter for details

Return:

- True in case of success
- False in case of failure

## Trace interface

---

### Trace verbosity levels

---

C++ trace levels are described in header file is located in <P7>/Headers/P7\_Trace.h

```
enum eP7Trace_Level
{
    EP7TRACE_LEVEL_TRACE           = 0,
    EP7TRACE_LEVEL_DEBUG           ,
    EP7TRACE_LEVEL_INFO            ,
    EP7TRACE_LEVEL_WARNING         ,
    EP7TRACE_LEVEL_ERROR           ,
    EP7TRACE_LEVEL_CRITICAL        ,
    EP7TRACE_LEVEL_COUNT           ,
};
```

C trace levels are described in header file is located in <P7>/Headers/P7\_Cproxy.h

```
#define P7_TRACE_LEVEL_TRACE      0
#define P7_TRACE_LEVEL_DEBUG      1
#define P7_TRACE_LEVEL_INFO       2
#define P7_TRACE_LEVEL_WARNING    3
#define P7_TRACE_LEVEL_ERROR      4
#define P7_TRACE_LEVEL_CRITICAL   5
```

### Trace format string

---

C++/C interfaces supports variable arguments format string, like `Trace("Value = %d, %08x", 10, 20)`.

Next field characters are supported:

- type field characters: l, I, h, w, c, C, d, i, o, u, x, X, p, n, S, s, e, E, f, g, G, a, A
- prefixes and format-Type specifiers: l64, l32, ll, l, h, l, w

Full documentation about format string you can find here: <http://msdn.microsoft.com/en-us/library/56e442dc.aspx>

## C++ interface

Trace header file is located in <P7>/Headers/P7\_Trace.h

### P7\_Create\_Trace

Function allows to create `IP7_Trace` object

```
IP7_Trace* P7_Create_Trace(IP7_Client *i_pClient, const tXCHAR *i_pName)
```

*Parameters:*

- `i_pClient` – pointer to client object
- `i_pName` – name of the trace channel

*Return:*

- Valid pointer to `IP7_Trace` object in case of success
- NULL in case of failure

### P7\_Get\_Shared\_Trace

This functions allows you to get P7 trace instance if it was created by someone else inside current process and shared using `IP7_Trace::Share(...)` function.

Sharing mechanism is very flexible way to redistribute your `IP7_Trace` object among your modules without passing pointer to it and modification your interfaces, function is thread safe.

```
IP7_Trace *P7_Get_Shared_Trace(const tXCHAR *i_pName)
```

*Parameters:* name of previously shared P7 trace instance

*Return:*

- Valid pointer to `IP7_Trace` interface in case of success
- NULL in case of failure

N.B.: Every successful call of this function increase reference counter value on retrieved `IP7_Trace` object, **do not forget to call `Release()` function**

### IP7\_Trace::Add\_Ref

Function increase object reference counter

```
tINT32 Add_Ref()
```

*Return:* object's reference counter new value

### IP7\_Trace::Release

Function decrease object reference counter, object will be destroyed when reference counter is equal to 0.

```
tINT32 Release()
```

*Return:* object's reference counter new value

### *IP7\_Trace::Register\_Thread*

Function register thread name using it ID, used to match later on Baical side thread ID and human readable thread name. Call this function when new thread is created and do not forget to call `Unregister_Thread` when thread has to be closed.

```
tBOOL Register_Thread(const tXCHAR *i_pName, tUINT32 i_dwThreadId)
```

*Parameters:*

- `i_pName` – thread name
- `i_dwThreadId` – ID of the thread, if `i_dwThreadId == 0` then current thread ID will be used.

*Return:*

- TRUE – success
- FALSE – failure

### *IP7\_Trace::Unregister\_Thread*

Function unregister thread, used to match later on Baical side thread ID and human readable thread name.

```
tBOOL Unregister_Thread(tUINT32 i_dwThreadId)
```

*Parameters:*

- `i_dwThreadId` – ID of the thread, if `i_dwThreadId == 0` then current thread ID will be used.

*Return:*

- TRUE – success
- FALSE – failure

### *IP7\_Trace::Register\_Module*

Function register application module. If application or library which uses P7 contains different parts (modular architecture) you may use this function. It allows you:

- To have nice output on Baical side, in addition to module ID – module name will be printed for every trace message
- Independent verbosity level management for every module. Module verbosity may be set online through Baical.

Usage of this function does not have an impact on performance of traces, modules information are transmitted only once.

```
tBOOL Register_Module(const tXCHAR *i_pName, IP7_Trace::hModule *o_hModule)
```

*Parameters:*

- `i_pName` – module name (case sensitive), if module with the same name is already exist – handle to that module will be returned
- `o_pModule` – module handle (output).

*Return:*

- TRUE – success
- FALSE – failure

### *IP7\_Trace::Set\_Verbosity*

Function sets trace channel verbosity level, all traces with less priority will be rejected, you may set verbosity level on-line from Baical server.

Verbosity levels are described in chapter [Trace verbosity levels](#).

```
void Set_Verbosity(IP7_Trace::hModule i_hModule, eP7Trace_Level i_eVerbosity)
```

*Parameters:*

- `i_hModule` – module handle, if handle is NULL global verbosity will be set for whole P7.Trace object
- `i_eVerbosity` – [trace verbosity levels](#).

### *IP7\_Trace::Share*

Function allows to share current P7 trace instance in address space of current process. Sharing mechanism is very flexible way to redistribute your `IP7_Trace` object among your modules without passing pointer to it and modification your interfaces, function is thread safe.

```
tBOOL Share(const tXCHAR *i_pName)
```

*Parameters:* name of shared P7 client instance, should be unique

*Return:*

- TRUE – success
- FALSE – failure, the other object with the same name is already shared inside current process

### *IP7\_Trace::Trace*

Function sent trace message, it has variable arguments list.

```
tBOOL Trace(tUINT16 i_wTrace_ID,
            eP7Trace_Level i_eLevel,
            IP7_Trace::hModule i_hModule,
            tUINT16 i_wLine,
            const char *i_pFile,
            const char *i_pFunction,
            const tXCHAR *i_pFormat,
            ...
)
```

*Parameters:*

- `i_wTrace_ID` – hardcoded trace ID, possible range is [0 .. 1023]. This ID is used to match trace data and trace format string on server side. You can specify this parameter in range [1..1023] if you want to send a trace as quickly as possible. Otherwise you can put 0 - and this function will work a little bit slowly, and ID will be auto-calculated
- `i_eLevel` – trace level (error, warning, debug, etc). Described in chapter [Trace verbosity levels](#)
- `i_hModule` – module handle, it is useful for further filtering on Baical side, may be NULL
- `i_wLine` – source file line number from where your trace is called (C/C++ preprocessor macro `__LINE__`)
- `i_pFile` – source file line number from where your trace is called (C/C++ preprocessor macro `__FILE__`)
- `i_pFunction` – source file name from where your trace is called. (C/C++ preprocessor macro `__FUNCTION__`)
- `i_pFormat` – format string (like "Value = %d, %08x"). Described in chapter [Trace format string](#)
- ... - variable arguments

Return:

- TRUE – success
- FALSE – failure, there are few possible reasons for failure (for details see logs):
  - No free buffers to store new trace, P7 client do not have enough time to deliver all trace/telemetry messages and there is no free buffers
  - Baical server is not available (if Sink is Baical)
  - No free space on HDD (if Sink is file)

**N.B.:** DO NOT USE VARIABLES for format string, file name, function name! You should always use CONSTANT TEXT like "My Format %d, %s", "myfile.cpp", "myfunction"

To simplify function call you may use macro defined in <P7>/Headers/P7\_Trace.h:

- P7\_TRACE
- P7\_DEBUG
- P7\_INFO
- P7\_WARNING
- P7\_ERROR
- P7\_CRITICAL

### *IP7\_Trace::Trace\_Embedded*

Function is similar to Trace(...) function, but intended for embedding into existing logging/trace function inside your code.

```
tBOOL Trace_Embedded(tUINT16          i_wTrace_ID,
                    eP7Trace_Level  i_eLevel,
                    IP7_Trace::hModule i_hModule,
                    tUINT16          i_wLine,
                    const char       *i_pFile,
                    const char       *i_pFunction,
                    const tXCHAR      **i_ppFormat
                    )
```

Parameters:

- i\_wTrace\_ID – hardcoded trace ID, possible range is [0 .. 1023]. This ID is used to match trace data and trace format string on server side. You can specify this parameter in range [1..1023] if you want to send a trace as quickly as possible. Otherwise you can put 0 - and this function will work a little bit slowly, and ID will be auto-calculated
- i\_eLevel – trace level (error, warning, debug, etc). Described in chapter [Trace verbosity levels](#)
- i\_hModule – module handle, it is useful for further filtering on Baical side, may be NULL
- i\_wLine – source file line number from where your trace is called (C/C++ preprocessor macro `__LINE__`)
- i\_pFile – source file line number from where your trace is called (C/C++ preprocessor macro `__FILE__`)
- i\_pFunction – source file name from where your trace is called. (C/C++ preprocessor macro `__FUNCTION__`)
- i\_ppFormat – address of format string (like "Value = %d, %08x"). Described in chapter [Trace format string](#)

Return:

- TRUE – success

- FALSE – failure, there are few possible reasons for failure (for details see logs):
  - No free buffers to store new trace, P7 client do not have enough time to deliver all trace/telemetry messages and there is no free buffers
  - Baical server is not available (if Sink is Baical)
  - No free space on HDD (if Sink is file)

**N.B.:** DO NOT USE VARIABLES for format string, file name, function name! You should always use CONSTANT TEXT like "My Format %d, %s", "myfile.cpp", "myfunction"

### *IP7\_Trace::Trace\_Managed*

Function is similar to Trace(...) function, but intended for usage with managed languages like C#, python, VB, etc. It is not so efficient like Trace() or Trace\_Embedded() functions (about 25% less efficient)

```
tBOOL Trace_Managed(tUINT16          i_wTrace_ID,
                   eP7Trace_Level  i_eLevel,
                   IP7_Trace::hModule i_hModule,
                   tUINT16          i_wLine,
                   const tXCHAR      *i_pFile,
                   const tXCHAR      *i_pFunction,
                   const tXCHAR      *i_pMessage
                   )
```

#### *Parameters:*

- i\_wTrace\_ID – hardcoded trace ID, possible range is [0 .. 1023]. This ID is used to match trace data and trace format string on server side. You can specify this parameter in range [1..1023] if you want to send a trace as quickly as possible. Otherwise you can put 0 - and this function will work a little bit slowly, and ID will be auto-calculated
- i\_eLevel – trace level (error, warning, debug, etc). Described in chapter [Trace verbosity levels](#)
- i\_hModule – module handle, it is useful for further filtering on Baical side, may be NULL
- i\_wLine – source file line number from where your trace is called (C/C++ preprocessor macro `__LINE__`)
- i\_pFile – source file line number from where your trace is called (C/C++ preprocessor macro `__FILE__`)
- i\_pFunction – source file name from where your trace is called. (C/C++ preprocessor macro `__FUNCTION__`)
- i\_pMessage – trace text message

#### *Return:*

- TRUE – success
- FALSE – failure, there are few possible reasons for failure (for details see logs):
  - No free buffers to store new trace, P7 client do not have enough time to deliver all trace/telemetry messages and there is no free buffers
  - Baical server is not available (if Sink is Baical)
  - No free space on HDD (if Sink is file)

### **C interface**

Trace header file is located in <P7>/Headers/P7\_Cproxy.h

### *P7\_Trace\_Create*

Function allows to create P7 trace object

```
hP7_Trace P7_Trace_Create(hP7_Client i_hClient, const tXCHAR *i_pName)
```

*Parameters:*

- i\_pClient – client object handle
- i\_pName – name of the trace channel

*Return:*

- Valid handle of P7 trace object in case of success
- NULL in case of failure

### P7\_Trace\_Get\_Shared

This functions allows you to get P7 trace instance if it was created by someone else inside current process and shared using `P7_Trace_Share(...)` function.

Sharing mechanism is very flexible way to redistribute your `hP7_Trace` object among your modules without passing pointer to it and modification your interfaces, function is thread safe.

```
hP7_Trace __cdecl P7_Trace_Get_Shared(const tXCHAR *i_pName)
```

*Parameters:* name of previously shared P7 trace instance

*Return:*

- Valid `hP7_Trace` handle of P7 trace object
- NULL in case of failure

N.B.: Every successful call of this function increase reference counter value on retrieved object handle, **do not forget to call `P7_Trace_Release()` function**

### P7\_Trace\_Add\_Ref

Function increase object reference counter

```
tINT32 P7_Trace_Add_Ref(hP7_Trace i_hTrace)
```

*Parameters:*

- i\_hTrace – Trace object handle

*Return:* object's reference counter new value

### P7\_Trace\_Release

Function decrease object reference counter, object will be destroyed when reference counter is equal to 0.

```
tINT32 P7_Trace_Release(hP7_Trace i_hTrace)
```

*Parameters:*

- i\_hTrace – Trace object handle

*Return:* object's reference counter new value

### *P7\_Trace\_Register\_Thread*

Function register thread name using its ID, used to match later on Baical side thread ID and human readable thread name. Call this function when new thread is created and do not forget to call `P7_Trace_Unregister_Thread` when thread has to be closed.

```
tBOOL P7_Trace_Register_Thread(hP7_Trace i_hTrace, const txCHAR*i_pName, tUINT32 i_dwThreadId)
```

*Parameters:*

- `i_hTrace` – Trace object handle
- `i_pName` – thread name
- `i_dwThreadId` – ID of the thread, if `i_dwThreadId == 0` then current thread ID will be used.

*Return:*

- `TRUE` – success
- `FALSE` – failure

### *P7\_Trace\_Unregister\_Thread*

Function unregister thread, used to match later on Baical side thread ID and human readable thread name.

```
tBOOL P7_Trace_Unregister_Thread(hP7_Trace i_hTrace, tUINT32 i_dwThreadId)
```

*Parameters:*

- `i_hTrace` – Trace object handle
- `i_dwThreadId` – ID of the thread, if `i_dwThreadId == 0` then current thread ID will be used.

*Return:*

- `TRUE` – success
- `FALSE` – failure

### *P7\_Trace\_Register\_Module*

Function register application module. If application or library which uses P7 contains different parts (modular architecture) you may use this function. It allows you:

- To have nice output on Baical side, in addition to module ID – module name will be printed for every trace message
- Independent verbosity level management for every module. Module verbosity may be set online through Baical.

Usage of this function does not have an impact on performance of traces, modules information are transmitted only once.

```
hP7_Trace_Module P7_Trace_Register_Module(hP7_Trace i_hTrace, const txCHAR *i_pName)
```

*Parameters:*

- `i_hTrace` – Trace object handle
- `i_pName` – module name (case sensitive), if module with the same name already exists – handle to that module will be returned

*Return:*

- module handle

### *P7\_Trace\_Set\_Verbosity*

Function sets trace channel verbosity level, all traces with less priority will be rejected, you may set verbosity level on-line from Baical server.

```
void P7_Trace_Set_Verbosity(hP7_Trace i_hTrace,
                          hP7_Trace_Module i_hModule,
                          tUINT32 i_dwVerbosity
                          )
```

*Parameters:*

- i\_hTrace – trace object handle
- i\_hModule – module handle, if handle is NULL global verbosity will be set for whole P7.Trace object
- i\_dwVerbosity – verbosity level, next values are available:
  - P7\_TRACE\_LEVEL\_TRACE (0)
  - P7\_TRACE\_LEVEL\_DEBUG (1)
  - P7\_TRACE\_LEVEL\_INFO (2)
  - P7\_TRACE\_LEVEL\_WARNING (3)
  - P7\_TRACE\_LEVEL\_ERROR (4)
  - P7\_TRACE\_LEVEL\_CRITICAL (5)

### *P7\_Trace\_Share*

Function allows to share current client instance in address space of current process. Sharing mechanism is very flexible way to redistribute your P7 trace object among your modules without passing pointer to it and modification your interfaces, function is thread safe.

```
tBOOL P7_Trace_Share(hP7_Trace i_hTrace, const tXCHAR *i_pName)
```

*Parameters:*

- i\_hTrace – trace object handle
- i\_pName – name of shared P7 client instance, should be unique

*Return:*

- TRUE - success
- FALSE – failure, the other object with the same name is already shared inside current process

### *P7\_Trace\_Add*

Function sent trace message, it has variable arguments list.

```
tBOOL P7_Trace_Add(hP7_Trace i_hTrace,
                  tUINT16 i_wTrace_ID,
                  tUINT32 i_dwLevel,
                  hP7_Trace_Module i_hModule,
                  tUINT16 i_wLine,
                  const char *i_pFile,
                  const char *i_pFunction,
                  const tXCHAR *i_pFormat,
                  ...
                  )
```

*Parameters:*

- `i_hTrace` – trace object handle
- `i_wTrace_ID` – hardcoded trace ID, possible range is [0 .. 1023]. This ID is used to match trace data and trace format string on server side. You can specify this parameter in range [1..1023] if you want to send a trace as quickly as possible. Otherwise you can put 0 - and this function will work a little bit slowly, and ID will be auto-calculated
- `i_eLevel` – trace level (error, warning, debug, etc). Described in chapter [Trace verbosity levels](#)
- `i_hModule` – module handle, it is useful for further filtering on Baical side, may be NULL
- `i_wLine` – source file line number from where your trace is called (C/C++ preprocessor macro `__LINE__`)
- `i_pFile` – source file line number from where your trace is called (C/C++ preprocessor macro `__FILE__`)
- `i_pFunction` – source file name from where your trace is called. (C/C++ preprocessor macro `__FUNCTION__`)
- `i_pFormat` – format string (like "Value = %d, %08x"). Described in chapter [Trace format string](#)
- ... - variable arguments

*Return:*

- TRUE - success
- FALSE – failure, there are few possible reasons for failure (for details see logs):
  - No free buffers to store new trace, P7 client do not have enough time to deliver all trace/telemetry messages and there is no free buffers
  - Baical server is not available (if Sink is Baical)
  - No free space on HDD (if Sink is file)

**N.B.:** DO NOT USE VARIABLES for format string, file name, function name! You should always use CONSTANT TEXT like "My Format %d, %s", "myfile.cpp", "myfunction"

To simplify function call you may use macro `P7_TRACE_ADD` defined in `<P7>/Headers/P7_Cproxy.h`.

### P7\_Trace\_Embedded

Function is similar to `P7_Trace_Add(...)` function, but intended for embedding into existing logging/trace function inside your code.

```
tBOOL P7_Trace_Embedded(hP7_Trace    i_hTrace,
                       tUINT16      i_wTrace_ID,
                       tUINT32      i_dwLevel,
                       hP7_Trace_Module i_hModule,
                       tUINT16      i_wLine,
                       const char    *i_pFile,
                       const char    *i_pFunction,
                       const tXCHAR  **i_ppFormat
                       )
```

*Parameters:*

- `i_hTrace` – trace object handle
- `i_wTrace_ID` – hardcoded trace ID, possible range is [0 .. 1023]. This ID is used to match trace data and trace format string on server side. You can specify this parameter in range [1..1023] if you want to send a trace as quickly as possible. Otherwise you can put 0 - and this function will work a little bit slowly, and ID will be auto-calculated
- `i_eLevel` – trace level (error, warning, debug, etc). Described in chapter [Trace verbosity levels](#)
- `i_hModule` – module handle, it is useful for further filtering on Baical side, may be NULL
- `i_wLine` – source file line number from where your trace is called (C/C++ preprocessor macro `__LINE__`)

- `i_pFile` – source file line number from where your trace is called (C/C++ preprocessor macro `__FILE__`)
- `i_pFunction` – source file name from where your trace is called. (C/C++ preprocessor macro `__FUNCTION__`)
- `i_ppFormat` – address of format string (like "Value = %d, %08x"). Described in chapter [Trace format string](#)

*Return:*

- TRUE - success
- FALSE – failure, there are few possible reasons for failure (for details see logs):
  - No free buffers to store new trace, P7 client do not have enough time to deliver all trace/telemetry messages and there is no free buffers
  - Baical server is not available (if Sink is Baical)
  - No free space on HDD (if Sink is file)

**N.B.:** DO NOT USE VARIABLES for format string, file name, function name! You should always use CONSTANT TEXT like "My Format %d, %s", "myfile.cpp", "myfunction"

### P7\_Trace\_Managed

Function is similar to `P7_Trace_Add (...)` function, but intended for usage with managed languages like C#, python, VB, etc. It is not so efficient like `P7_Trace_Add ()` or `P7_Trace_Embedded()` functions (about 25% less efficient)

```
tBOOL P7_Trace_Managed(hP7_Trace    i_hTrace,
                      tUINT16      i_wTrace_ID,
                      tUINT32      i_dwLevel,
                      hP7_Trace_Module i_hModule,
                      tUINT16      i_wLine,
                      const tXCHAR  *i_pFile,
                      const tXCHAR  *i_pFunction,
                      const tXCHAR  *i_pMessage
                      )
```

*Parameters:*

- `i_hTrace` – trace object handle
- `i_wTrace_ID` – hardcoded trace ID, possible range is [0 .. 1023]. This ID is used to match trace data and trace format string on server side. You can specify this parameter in range [1..1023] if you want to send a trace as quickly as possible. Otherwise you can put 0 - and this function will work a little bit slowly, and ID will be auto-calculated
- `i_eLevel` – trace level (error, warning, debug, etc). Described in chapter [Trace verbosity levels](#)
- `i_hModule` – module handle, it is useful for further filtering on Baical side, may be NULL
- `i_wLine` – source file line number from where your trace is called (C/C++ preprocessor macro `__LINE__`)
- `i_pFile` – source file line number from where your trace is called (C/C++ preprocessor macro `__FILE__`)
- `i_pFunction` – source file name from where your trace is called. (C/C++ preprocessor macro `__FUNCTION__`)
- `i_pMessage` – trace text message

*Return:*

- TRUE - success
- FALSE – failure, there are few possible reasons for failure (for details see logs):

- No free buffers to store new trace, P7 client do not have enough time to deliver all trace/telemetry messages and there is no free buffers
- Baical server is not available (if Sink is Baical)
- No free space on HDD (if Sink is file)

## C# interface

---

C# shell file is located in <P7>/ Wrappers/C#/P7.cs

C# shell depending on P7x64.dll/ P7x32.dll you may generate them by building P7 solution

## P7.Traces

---

Constructor allows to create P7 trace object

```
P7.Traces Traces(P7.Client i_pClient, String i_sName)
```

Parameters:

- i\_pClient – client object class
- i\_pName – name of the trace channel

Return:

- P7 trace object in case of success
- ArgumentException(...) or ArgumentNullException(...) in case of failure

## P7.Traces.Get\_Shared

---

This functions allows you to get P7 trace instance if it was created by someone else inside current process and shared using `P7::Traces::Share(...)` function.

Sharing mechanism is very flexible way to redistribute your P7 trace object among your modules without passing pointer to it and modification your interfaces, function is thread safe.

```
static P7.Traces Get_Shared(String i_sName)
```

Parameters: name of previously shared P7 trace instance

Return:

- P7 trace object in case of success
- null in case of failure

## P7.Traces.Add\_Ref

---

Function increase object reference counter

```
System.Int32 AddRef()
```

Return: object's reference counter new value

### *P7.Trace.Release*

Function decrease object reference counter, object will be destroyed when reference counter is equal to 0.

```
System.IntPtr Release()
```

*Return:* object's reference counter new value

### *P7.Traces.Register\_Thread*

Function register thread name using it ID, used to match later on Baical side thread ID and human readable thread name. Call this function when new thread is created and do not forget to call `P7_Trace_Unregister_Thread` when thread has to be closed.

```
bool Register_Thread(String i_sName, UInt32 i_dwThreadID = 0)
```

*Parameters:*

- `i_sName` – thread name
- `i_dwThread_ID` – ID of the thread, if `i_dwThread_ID == 0` then current thread ID will be used.

*Return:*

- `true` – success
- `false` – failure

### *P7.Traces.Unregister\_Thread*

Function unregister thread, used to match later on Baical side thread ID and human readable thread name.

```
bool Unregister_Thread(UInt32 i_dwThreadID = 0)
```

*Parameters:*

- `i_dwThread_ID` – ID of the thread, if `i_dwThread_ID == 0` then current thread ID will be used.

*Return:*

- `TRUE` – success
- `FALSE` – failure

### *P7.Traces.Register\_Module*

Function register application module. If application or library which uses P7 contains different parts (modular architecture) you may use this function. It allows you:

- To have nice output on Baical side, in addition to module ID – module name will be printed for every trace message
- Independent verbosity level management for every module. Module verbosity may be set online through Baical.

Usage of this function does not have an impact on performance of traces, modules information are transmitted only once.

```
System.IntPtr Register_Module(String i_sName)
```

*Parameters:*

- `i_pName` – module name (case sensitive), if module with the same name is already exist – handle to that module will be returned

*Return:*

- module handle

***P7.Traces.Set\_Verbosity***

Function sets trace channel verbosity level, all traces with less priority will be rejected, you may set verbosity level on-line from Baical server.

```
void Set_Verbosity(System.IntPtr i_hModule, Traces.Level i_eLevel)
```

*Parameters:*

- `i_hModule` – module handle, if handle is null global verbosity will be set for whole P7.Trace object. To obtain module handle use `P7.Traces.Register_Module()` function
- `i_dwVerbosity` – verbosity level

***P7.Traces.Share***

Function allows to share current client instance in address space of current process. Sharing mechanism is very flexible way to redistribute your P7 trace object among your modules without passing pointer to it and modification your interfaces, function is thread safe.

```
bool Share(String i_sName)
```

*Parameters:*

- `i_sName` – name of shared P7 client instance, should be unique

*Return:*

- TRUE - success
- FALSE – failure, the other object with the same name is already shared inside current process

***P7.Traces.Trace***

Functions sent trace/debug/warning/error/critical messages.

```
bool Trace (System.IntPtr i_hModule, String i_sMessage)
bool Debug (System.IntPtr i_hModule, String i_sMessage)
bool Info (System.IntPtr i_hModule, String i_sMessage)
bool Warning (System.IntPtr i_hModule, String i_sMessage)
bool Error (System.IntPtr i_hModule, String i_sMessage)
bool Critical (System.IntPtr i_hModule, String i_sMessage)
```

*Parameters:*

- `i_hModule` – module handle, it is useful for further filtering on Baical side. To obtain module handle use `P7.Traces.Register_Module()` function
- `i_sMessage` – trace message

*Return:*

- TRUE - success

- FALSE – failure, there are few possible reasons for failure (for details see logs):
  - No free buffers to store new trace, P7 client do not have enough time to deliver all trace/telemetry messages and there is no free buffers
  - Baical server is not available (if Sink is Baical)
  - No free space on HDD (if Sink is file)

## Python interface

Python shell file is located in <P7>/ Wrappers/Py/P7.py

Python shell depending on P7x64.dll/P7x32.dll (Windows) and libP7.so (Linux) you may generate them by building P7 solution under Windows or run build.sh under Linux

## Importing

Importing is described in [Client Importing](#) chapter.

## P7. Get\_Trace\_Channel

Function allows to get P7 trace instance by name or create new one.

```
P7.Traces P7.Get_Trace_Channel(i_sTraceName, i_sClientName = None)
```

Parameters:

- i\_sTraceName – name of P7 trace instance, should be unique, used for sharing P7 trace in address space of current python session
- i\_sClientName – name of the client, registered by function call [Register\\_Client\(\)](#), may be empty if client with i\_sTraceName is already exists, otherwise i\_sClientName is used to create new trace object

Return:

- True in case of success
- None in case of failure

## P7. Traces.Enable\_Stack\_Info

Function allows enable/disable collecting stack information (file name & line, function name). Extracting information about stack takes a lot of time, about additional 500 microseconds on modern PC (2014). If you disable stack information - trace functions will be accelerated about 50-150 times depending on python version.

```
P7.Traces.Enable_Stack_Info(self, i_bEnabled)
```

Parameters:

- i\_bEnabled – disable – False, enable - True

## P7.Traces.Trace

Functions sent trace/debug/info/warning/error/critical messages.

```
P7.Traces.Trace(self, i_hModule, i_sMessage, i_bUseStackInfo = True)
P7.Traces.Debug(self, i_hModule, i_sMessage, i_bUseStackInfo = True)
```

```
P7.Traces.Info(self, i_hModule, i_sMessage, i_bUseStackInfo = True)
P7.Traces.Warning(self, i_hModule, i_sMessage, i_bUseStackInfo = True)
P7.Traces.Error(self, i_hModule, i_sMessage, i_bUseStackInfo = True)
P7.Traces.Critical(self, i_hModule, i_sMessage, i_bUseStackInfo = True)
```

#### Parameters:

- `i_hModule` – module handle, if handle is 0 global verbosity will be set for whole P7.Trace object. To obtain module handle use `P7.Traces.Register_Module()` function
- `i_sMessage` – trace message
- `i_bUseStackInfo` – enable or disable stack information, see [P7.Traces.Enable Stack Info](#) for details

#### Return:

- True – success
- False – failure, there are few possible reasons for failure (for details see logs):
  - No free buffers to store new trace, P7 client do not have enough time to deliver all trace/telemetry messages and there is no free buffers
  - Baical server is not available (if Sink is Baical)
  - No free space on HDD (if Sink is file)

### ***P7.Traces.Set\_Verbosity***

Function sets trace channel verbosity level, all traces with less priority will be rejected, you may set verbosity level on-line from Baical server.

```
P7.Traces.Set_Verbosity(self, i_hModule, i_iLevel)
```

#### Parameters:

- `i_hModule` – module handle, if handle is null global verbosity will be set for whole P7.Trace object. To obtain module handle use `P7.Traces.Register_Module()` function
- `i_iLevel` – verbosity level, there are next verbosity levels:
  - `P7.Traces.m_iTrace` (0)
  - `P7.Traces.m_iDebug` (1)
  - `P7.Traces.m_iInfo` (2)
  - `P7.Traces.m_iWarning` (3)
  - `P7.Traces.m_iError` (4)
  - `P7.Traces.m_iCritical` (5)

### ***P7.Traces.Register\_Module***

Function register application module. If application or library which uses P7 contains different parts (modular architecture) you may use this function. It allows you:

- To have nice output on Baical side, in addition to module ID – module name will be printed for every trace message
- Independent verbosity level management for every module. Module verbosity may be set online through Baical.

Usage of this function does not have an impact on performance of traces, modules information are transmitted only once.

```
hModule P7.Traces.Register_Module(self, i_sName)
```

*Parameters:*

- `i_pName` – module name (case sensitive), if module with the same name is already exist – handle to that module will be returned

*Return:*

- module handle (digit)

## Telemetry interface

### Configuration

For fine configuration and controlling of telemetry channel special structure is defined:

```
typedef void (__cdecl *fnTelemetry_Enable)(void *i_pContext, tUINT8 i_bId, tBOOL i_bEnable);
typedef tUINT64 (__cdecl *fnGet_Time_Stamp)(void *i_pContext);
typedef void (__cdecl *fnConnect)(void *i_pContext, tBOOL i_bConnected);

typedef struct
{
    void *pContext;
    tUINT64 qwTimestamp_Frequency;
    fnGet_Time_Stamp pTimestamp_Callback;
    fnTelemetry_Enable pEnable_Callback;
    fnConnect pConnect_Callback;
} stTelemetry_Conf;
```

Parameters:

- pContext – used defined context pointer, will be used with all callbacks
- qwTimestamp\_Frequency – in most of the cases telemetry channel uses hi precision system timestamps, but if you want to use more precise time stamp please fill this field with your time precision in Hz. This parameter has to be used only together with pTimestamp\_Callback function. Separate usage isn't allowed. Put 0 to use default system timestamp.
- pTimestamp\_Callback – call back to retrieve current user defined timestamp, will be called for **every** telemetry sample – so function should not bring performance penalties. Put **NULL** to use default system timestamp.
- pEnable\_Callback – call back function to be called when state of the counter has been changed (ON/OFF) remotely from Baical. **NULL** is default value
- pConnect\_Callback – call back function to be called when connection state has been changed. **NULL** is default value

fnTelemetry\_Enable function parameters:

- i\_pContext – context passed to stTelemetry\_Conf structure
- i\_bId – counter's ID
- i\_bEnable – counter's state TRUE = ON, FALSE = OFF

fnGet\_Time\_Stamp function parameters:

- i\_pContext – context passed to stTelemetry\_Conf structure

*Return:* timestamp value, 64 bits

fnConnect function parameters:

- i\_pContext – context passed to stTelemetry\_Conf structure
- i\_bConnect – connection state TRUE = ON, FALSE = OFF

### C++ interface

Trace header file is located in <P7>/Headers/P7\_Telemetry.h

### *P7\_Create\_Telemetry*

Function allows to create P7 telemetry object

```
IP7_Telemetry* P7_Create_Telemetry(IP7_Client *i_pClient,
                                   const tXCHAR *i_pName,
                                   const stTelemetry_Conf *i_pConf = NULL
                                   )
```

*Parameters:*

- `i_pClient` – pointer to client object
- `i_pName` – name of the telemetry channel
- `i_pConf` – telemetry channel [configuration](#), use NULL for default values.

*Return:*

- Valid pointer to [IP7\\_Telemetry](#) object in case of success
- NULL in case of failure

### *P7\_Get\_Shared\_Telemetry*

This functions allows you to get P7 telemetry instance if it was created by someone else inside current process and shared using [IP7\\_Telemetry::Share\(...\)](#) function.

Sharing mechanism is very flexible way to redistribute your P7 telemetry object among your modules without passing pointer to it and modification your interfaces, [function is thread safe](#).

```
IP7_Telemetry* P7_Get_Shared_Telemetry(const tXCHAR *i_pName)
```

*Parameters:* name of previously shared P7 telemetry instance

*Return:*

- Valid pointer to [IP7\\_Telemetry](#) interface in case of success
- NULL in case of failure

N.B.: Every successful call of this function increase reference counter value on retrieved [IP7\\_Telemetry](#) object, **do not forget to call [Release\(\)](#) function**

### *IP7\_Telemetry::Add\_Ref*

Function increase object reference counter

```
tINT32 Add_Ref()
```

*Return:* object's reference counter new value

### *IP7\_Telemetry::Release*

Function decrease object reference counter, object will be destroyed when reference counter is equal to 0.

```
tINT32 Release()
```

*Return:* object's reference counter new value

### *IP7\_Telemetry::Share*

Function allows to share current P7 telemetry instance in address space of current process. Sharing mechanism is very flexible way to redistribute your P7 telemetry object among your modules without passing pointer to it and modification your interfaces, function is thread safe.

```
tBOOL Share(const tXCHAR *i_pName)
```

*Parameters:* name of shared P7 telemetry instance, should be unique

*Return:*

- TRUE – success
- FALSE – failure, the other object with the same name is already shared inside current process

### *IP7\_Telemetry::Create*

Function creates new telemetry counter and return counter's ID. One channel can handle up to 256 independent counters.

```
tBOOL Create(const tXCHAR *i_pName,
             tINT64 i_llMin,
             tINT64 i_llMax,
             tINT64 i_llAlarm,
             tUINT8 i_bOn,
             tUINT8 *o_pID
            )
```

*Parameters:*

- i\_pName - name of counter, max length 64 characters, should be unique for current channel (case sensitive)
- i\_llMin – minimal counter value, helping information for visualization, later you can override it in telemetry viewer
- i\_llMax – maximal counter value, helping information for visualization, later you can override it in telemetry viewer
- i\_llAlarm – alarm counter value, helping information for visualization
- i\_bOn – parameter specifies is counter enabled (1) or disabled (0) by default, later you can enable/disable it in real time from Baical server.
- o\_pID – output parameter, receives ID of the counter, this value is used to add samples to the counter

*Return:*

- TRUE – in case of success
- FALSE – in case of failure, there are few possible reasons:
  - No empty counters, all 256 slots are busy
  - Not valid input parameters
  - Counters name is already used

### *IP7\_Telemetry::Add*

Function allows to add counter's sample.

```
tBOOL Add(tUINT8 i_bID, tINT64 i_llValue)
```

*Parameters:*

- i\_bID – counter ID
- i\_lIValue – sample value

Return:

- TRUE – in case of success
- FALSE – in case of failure, there are few possible reasons:
  - No network connection (if Sink=Baical)
  - No free HDD space (if Sink=File)
  - Not valid input parameters

### *IP7\_Telemetry::Find*

Function finds counter's ID by its name. Search is case sensitive.

```
tBOOL Find(const tXCHAR *i_pName, tUINT8 *o_pID)
```

Parameters:

- i\_pName - name of counter
- o\_pID – output parameter, receives ID of the counter, this value is used to add samples to the counter

Return:

- TRUE – success, counter is found
- FALSE – failure, no counter with such name

### **C interface**

Trace header file is located in <P7>/Headers/P7\_Cproxy.h

### *P7\_Telemetry\_Create*

Function allows to create P7 telemetry object

```
hP7_Telemetry P7_Telemetry_Create(hP7_Client i_hClient,
                                  const tXCHAR *i_pName,
                                  const stTelemetry_Conf *i_pConf
                                  )
```

Parameters:

- i\_hClient – client object handle
- i\_pName – name of the telemetry channel
- i\_pConf – telemetry channel [configuration](#), use NULL for default values.

Return:

- Valid handle of P7 telemetry object in case of success
- NULL in case of failure

### *P7\_Telemetry\_Get\_Shared*

This functions allows you to get P7 telemetry instance if it was created by someone else inside current process and shared using *P7\_Telemetry\_Share(...)* function.

Sharing mechanism is very flexible way to redistribute your P7 telemetry object among your modules without passing pointer to it and modification your interfaces, function is thread safe.

```
tBOOL P7_Telemetry_Share(hP7_Telemetry i_hTelemetry, const tXCHAR *i_pName)
```

*Parameters:* name of previously shared P7 telemetry instance

*Return:*

- Valid handle of P7 telemetry object in case of success
- NULL in case of failure

N.B.: Every successful call of this function increase reference counter value on retrieved P7 telemetry object, **do not forget to call `P7_Telemetry_Release()` function**

### *P7\_Telemetry\_Add\_Ref*

Function increase object reference counter

```
tINT32 P7_Telemetry_Add_Ref(hP7_Telemetry i_hTelemetry)
```

*Parameters:* P7 telemetry handle

*Return:* object's reference counter new value

### *P7\_Telemetry\_Release*

Function decrease object reference counter, object will be destroyed when reference counter is equal to 0.

```
tINT32 P7_Telemetry_Release(hP7_Telemetry i_hTelemetry)
```

*Parameters:* P7 telemetry handle

*Return:* object's reference counter new value

### *P7\_Telemetry\_Share*

Function allows to share current P7 telemetry instance in address space of current process. Sharing mechanism is very flexible way to redistribute your P7 telemetry object among your modules without passing pointer to it and modification your interfaces, function is thread safe.

```
tBOOL P7_Telemetry_Share(hP7_Telemetry i_hTelemetry, const tXCHAR *i_pName)
```

*Parameters:*

- `i_hTelemetry` – P7 telemetry object handle
- `i_pName` - name of shared P7 telemetry instance, should be unique

*Return:*

- TRUE – success
- FALSE – failure, the other object with the same name is already shared inside current process

### P7\_Tlemetry\_Create\_Counter

Function create new telemetry counter and return counter's ID, one channel can handle up to 256 independent counters.

```
tBOOL P7_Tlemetry_Create_Counter(hP7_Tlemetry i_hTelemetry,
                                const tXCHAR *i_pName,
                                tINT64      i_llMin,
                                tINT64      i_llMax,
                                tINT64      i_llAlarm,
                                tUINT8      i_bOn,
                                tUINT8      *o_pCounter_ID
                                )
```

#### Parameters:

- i\_hTelemetry – P7 telemetry object handle
- i\_pName - name of counter, max length 64 characters, should be unique for current channel (case sensitive)
- i\_llMin – minimal counter value, helping information for visualization, later you can override it in telemetry viewer
- i\_llMax – maximal counter value, helping information for visualization, later you can override it in telemetry viewer
- i\_llAlarm – alarm counter value, helping information for visualization
- i\_bOn – parameter specifies is counter enabled (1) or disabled (0) by default, later you can enable/disable it in real time from Baical server.
- o\_pID – output parameter, receives ID of the counter, this value is used to add samples to the counter

#### Return:

- TRUE – in case of success
- FALSE – in case of failure, there are few possible reasons:
  - No empty counters, all 256 slots are busy
  - Not valid input parameters
  - Counters name is already used

### P7\_Tlemetry\_Put\_Value

Function allows to add counter's sample.

```
tBOOL P7_Tlemetry_Put_Value(hP7_Tlemetry i_hTelemetry,
                             tUINT8      i_bCounter_ID,
                             tINT64      i_llValue
                             )
```

#### Parameters:

- i\_hTelemetry – P7 telemetry object handle
- i\_bID – counter ID
- i\_llValue – sample value

#### Return:

- TRUE – in case of success
- FALSE – in case of failure, there are few possible reasons:
  - No network connection (if Sink=Baical)
  - No free HDD space (if Sink=File)
  - Not valid input parameters

### *P7\_Telemetry\_Find\_Counter*

Function finds counter's ID by its name. Search is case sensitive.

```
tBOOL P7_Telemetry_Find_Counter(hP7_Telemetry i_hTelemetry,
                               const tXCHAR *i_pName,
                               tUINT8      *o_pCounter_ID
                               )
```

*Parameters:*

- i\_hTelemetry – P7 telemetry object handle
- i\_pName - name of counter
- o\_pID – output parameter, receives ID of the counter, this value is used to add samples to the counter

*Return:*

- TRUE – success, counter is found
- FALSE – failure, no counter with such name

### *C# interface*

C# shell file is located in <P7>/ Wrappers/C#/P7.cs

C# shell depending on P7x64.dll/ P7x32.dll you may generate them by building P7 solution

### *P7.Telemetry*

Constructor allows to create P7 telemetry object

```
P7.Telemetry Telemetry(P7.Client i_pClient, String i_sName)
```

*Parameters:*

- i\_pClient – client object class
- i\_pName – name of the trace channel

*Return:*

- P7 trace object in case of success
- ArgumentException(...) or ArgumentNullException(...) in case of failure

### *P7.Telemetry.Get\_Shared*

This functions allows you to get P7 telemetry instance if it was created by someone else inside current process and shared using `P7::Telemetry::Share(...)` function.

Sharing mechanism is very flexible way to redistribute your P7 telemetry object among your modules without passing pointer to it and modification your interfaces, function is thread safe.

```
P7.Telemetry Get_Shared(String i_sName)
```

*Parameters:* name of previously shared P7 telemetry instance

*Return:*

- P7 trace object in case of success
- null in case of failure

### *P7.Telemetry.Add\_Ref*

Function increase object reference counter

```
System.Int32 AddRef()
```

*Return:* object's reference counter new value

### *P7.Telemetry.Release*

Function decrease object reference counter, object will be destroyed when reference counter is equal to 0.

```
System.Int32 Release()
```

*Return:* object's reference counter new value

### *P7.Telemetry.Share*

Function allows to share current client instance in address space of current process. Sharing mechanism is very flexible way to redistribute your P7 telemetry object among your modules without passing pointer to it and modification your interfaces, function is thread safe.

```
bool Share(String i_sName)
```

*Parameters:*

- *i\_sName* – name of shared P7 client instance, should be unique

*Return:*

- TRUE - success
- FALSE – failure, the other object with the same name is already shared inside current process

### *P7.Tlemetry.Create*

Function creates new telemetry counter and return counter's ID. One channel can handle up to 256 independent counters.

```
bool Create(String i_sName,
             System.Int64 i_llMin,
             System.Int64 i_llMax,
             System.Int64 i_llAlarm,
             System.Byte i_bOn,
             ref System.Byte o_rCounter_ID
            )
```

*Parameters:*

- *i\_sName* - name of counter, max length 64 characters, should be unique for current channel (case sensitive)
- *i\_llMin* – minimal counter value, helping information for visualization, later you can override it in telemetry viewer
- *i\_llMax* – maximal counter value, helping information for visualization, later you can override it in telemetry viewer
- *i\_llAlarm* – alarm counter value, helping information for visualization

- `i_bOn` – parameter specifies is counter enabled (1) or disabled (0) by default, later you can enable/disable it in real time from Baical server.
- `o_rCounter_ID` – output parameter, receives ID of the counter, this value is used to add samples to the counter

*Return:*

- TRUE – in case of success
- FALSE – in case of failure, there are few possible reasons:
  - No empty counters, all 256 slots are busy
  - Not valid input parameters
  - Counters name is already used

### *P7.Telemetry.Add*

Function allows to add counter's sample.

```
bool Add(System.Byte i_bCounter_ID, System.Int64 i_lValue)
```

*Parameters:*

- `i_bID` – counter ID
- `i_lValue` – sample value

*Return:*

- TRUE – in case of success
- FALSE – in case of failure, there are few possible reasons:
  - No network connection (if Sink=Baical)
  - No free HDD space (if Sink=File)
  - Not valid input parameters

### *P7.Telemetry.Find\_Counter*

Function finds counter's ID by its name. Search is case sensitive.

```
bool Find_Counter(String i_sName, ref System.Byte o_rCounter_ID)
```

*Parameters:*

- `i_sName` - name of counter
- `o_rCounter_ID` – output parameter, receives ID of the counter, this value is used to add samples to the counter

*Return:*

- TRUE – success, counter is found
- FALSE – failure, no counter with such name

### **Python interface**

Python shell file is located in <P7>/ Wrappers/Py/P7.py

Python shell depending on P7x64.dll/P7x32.dll (Windows) and libP7.so (Linux) you may generate them by building P7 solution under Windows or run build.sh under Linux

## Importing

Importing is described in [Client Importing](#) chapter.

### P7. Get\_Telemetry\_Channel

Function allows to get P7 telemetry instance by name or create new one.

```
P7.Telemetry P7.Get_Telemetry_Channel(i_sTelemetryName, i_sClientName = None)
```

#### Parameters:

- `i_sTelemetryName` – name of P7 telemetry object, should be unique, used for sharing P7 telemetry object in address space of current python session
- `i_sClientName` – name of the client, registered by function call [Register\\_Client\(\)](#), may be empty if client with `i_sTraceName` is already exists, otherwise `i_sClientName` is used to create new trace object

#### Return:

- True in case of success
- None in case of failure

### P7. Telemetry.Create

Function creates new telemetry counter and return counter's ID. One channel can handle up to 256 independent counters.

```
byte P7.Telemetry.Create(self, i_sName, i_lMin, i_lMax, i_lAlarm, i_bOn)
```

#### Parameters:

- `i_sName` - name of counter, max length 64 characters, should be unique for current channel (case sensitive)
- `i_lMin` – minimal counter value (signed 64 bits), helping information for visualization, later you can override it in telemetry viewer
- `i_lMax` – maximal counter value (signed 64 bits), helping information for visualization, later you can override it in telemetry viewer
- `i_lAlarm` – alarm counter value (signed 64 bits), helping information for visualization
- `i_bOn` – parameter specifies is counter enabled (1) or disabled (0) by default, later you can enable/disable it in real time from Baical server.

#### Return:

- [0..256] – in case of success
- [-1] – in case of failure, there are few possible reasons:
  - No empty counters, all 256 slots are busy
  - Not valid input parameters
  - Counters name is already used

### P7.Telemetry.Add

Function allows to add counter's sample.

```
bool P7.Telemetry.Add(self, i_bId, i_lValue)
```

*Parameters:*

- `i_bID` – counter ID
- `i_IValue` – sample value

*Return:*

- `TRUE` – in case of success
- `FALSE` – in case of failure, there are few possible reasons:
  - No network connection (if Sink=Baical)
  - No free HDD space (if Sink=File)
  - Not valid input parameters

### ***P7.Telemetry.Find\_Counter***

---

Function finds counter's ID by its name. Search is case sensitive.

```
byte P7.Telemetry.Find_Counter(self, i_sName)
```

*Parameters:*

- `i_sName` - name of counter

*Return:*

- `[0..255]` – success, counter is found
- `[-1]` – failure, no counter with such name

## P7 instances redistributing

It is always difficult to integrate new library into your project especially if library functionality have to be called from many different projects parts. You have to update your internal interfaces to pass pointers, handles, classes or create new abstraction layer or even worse – create singleton.

P7 has mechanism (thread safe) to simplify integration process and gives you ability just use it without internal interfaces modification – sharing mechanism.

It pretty simple to use it, for example you have some place in your project where you are going to initialize P7 trace/telemetry instances:

```
//TRACE MODULE, create client & trace channel
IP7_Client l_hClient = P7_Create_Client(TM("/P7.Sink=Baical /P7.Addr=127.0.0.1"));
IP7_Trace l_hTrace = P7_Create_Trace(l_hClient, TM("TraceChannel"));

//share the trace instance with unique name
l_hTrace->Share(TM("MySharedTrace"));
```

And then from any place or your project you can do:

```
//ANY OTHER MODULE, getting shared instance
IP7_Trace l_hTrace = P7_Get_Shared_Trace(l_hClient, TM("MySharedTrace"));

//using it
if (l_hTrace)
{
    l_hTrace->P7_INFO(0, TM("Information message #%d"), 0);
    ...

    //release the instance
    l_hTrace->Release();
}
```

You may use sharing mechanism for P7 client, P7 telemetry or trace channels.

## Process crush handling

Sometimes your application is killed by exception which you can't handle (like access violation or segmentation fault for example) and if you are using P7 intensively – last part of the trace or telemetry data may be lost due to internal buffering mechanism.

Sometimes it is good to know what the last trace message from dying process was. To handle such situation you have to:

1. Intercept process crush. How to catch moment of your application/process crash you could read in those articles:
  - Windows:
    - <http://www.codeproject.com/Articles/207464/Exception-Handling-in-Visual-Cplusplus>
  - Linux:
    - <http://ru.scribd.com/doc/3726406/Crash-N-Burn-Writing-Linux-application-fault-handlers>
    - <http://www.linuxprogrammingblog.com/all-about-linux-signals?page=show>
2. From crush handler function you should call once next function: [P7 Exceptional Flush](#), (there are analogs of that function for C, C# and Python languages). This function will deliver the rest of the data staying in internal buffers.

## Compilation

Library has not external dependencies this is why compilation is very simple:

- Windows - open P7.sln in Visual Studio 2010 or newer, choose debug/release, x86/x64 and rebuild the solution, all binaries will be gathered into <P7 folder>/Binaries
- Linux – run build.sh shell script to build all binaries, or run `./build.sh /clean` to clean temporary files. All binaries will be gathered into <P7 folder>/Binaries

Generated binaries:

- P7 static library (P7xXX.lib/libP7.a)
- P7 dynamic library (P7xXX.dll/libP7.so)
- Examples applications
- Tests applications

## Shared library (dll, so) export functions

P7 shared library (P7x32.dll, P7x64.dll, libP7.so) exports functions are described in C interface:

- [Client interface](#)
- [Trace interface](#)
- [Telemetry interface](#)

In addition inside <P7>/Headers/P7\_Cproxy.h header file, function types definitions are described for every exported C function, for example:

```
////////////////////////////////////  
//P7_Client_Create - function creates new P7 client, client is used as transport  
//...  
extern hp7_client __cdecl P7_Client_Create(const tXCHAR *i_pArgs);  
  
//dll/so function prototype  
typedef hp7_client (__cdecl *fnP7_Client_Create)(const tXCHAR *i_pArgs);
```

How to load & use dynamic (shared) libraries is described in next articles (with examples):

- [http://msdn.microsoft.com/en-us/library/windows/desktop/ms686944\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms686944(v=vs.85).aspx)
- <http://linux.die.net/man/3/dlopen>

## Examples

There are few examples available on different languages, they are located in [<P7 folder>/ Examples](#).

Examples are available for next languages:

- C
- C++
- C#
- Python

Examples are pretty simple and they show how:

- create & initialize P7 client
- create & initialize trace and telemetry channels
- send trace messages & telemetry samples
- release library resources