µP7 library integrator manual

Contents

Introduction

µP7 (micro P7) is lightweight C library for sending trace/logs & telemetry data from your Micro-controller's firmware to host/HW FIFO/cycle buffer/network/etc. for further analysis.

It is designed to be integrated on almost every microcontroller, even with very limited resources.

The library is oriented on firmware developers (Bare-metal or RTOS) & real-time task.

 If you are looking for logging library for Linux/Windows we consider you to have a look to P7 library.

Basic facts:

- extremely low memory usage, (900 lines of code, no dynamic memory allocation, all traces/logs format strings are removed from compilation & binaries)
- speed is priority, library is designed to minimize time per trace/telemetry call, for example average performance for Intel i7-870 is:
  - 15 million traces per second
  - 23 million telemetry samples per second
- Unicode support (UTF-8, UTF-16, UTF-32)
- no external dependencies
- remote management from Baical server (set verbosity per module, enable/disable telemetry counters) is possible
- big/little endian support
- providing maximum information for every trace message:
  - trace/log message (format string + Variable arguments)
  - Function name, file name, file line number
  - Module ID & name (if it was registered)
  - Level (error, warning, .. etc.)
  - Time with max platform accuracy
  - Trace ID
  - Sequence Number
  - Current thread ID (if RTOS is used)
- pre-processor is used to limit memory & .BSS segment usage

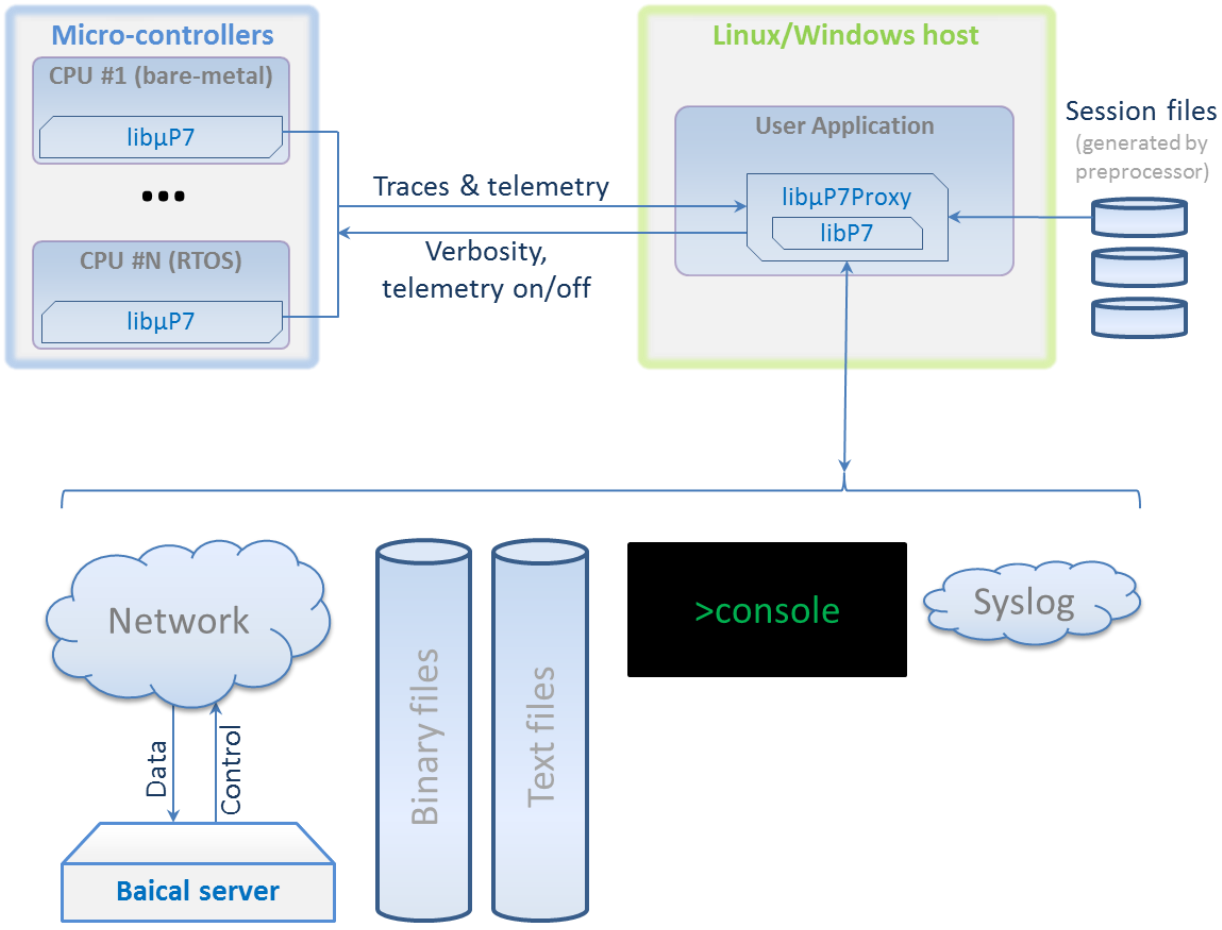
Directory structure

- Documentation (folder with library documentation)
- P7Lib (P7 library, it is used by proxy to send data to Baical server/file. **External P7 library may be used**)
- uP7Example (uP7 & uP7Proxy integration example)
- uP7Lib (uP7 library core)
  - Headers (uP7 API headers)
  - Platforms (uP7 platform headers)
  - Source (uP7 core)
- uP7preProcessor (uP7 preprocessor, tool to pre-process user's source files)
- uP7ProxyLib (uP7 proxy library, it is in charge of receiving uP7 packets and transform them to P7 protocol)
- uP7Test (uP7 tests, C++11 required)
- CMakeLists.txt – CMake (>= v3.8) script file

Design overview

µP7 consist of next components/tools:

- µP7 library (integrated into user firmware)
- µP7 proxy library (integrated into user host application, if any)
  - handles all µP7 incoming & outgoing data
  - decode it (using session files, generated by preprocessor)
  - transform it to appropriate shape for final user (sending to Baical server or syslog, saving to files)
- µP7 pre-processor
  - pre-process user C/C++ files, making them compatible with µP7 library
  - generate session files, to be used later by µP7 proxy library. Session files contain all necessary information to decode µP7 traffic.



µP7 is lightweight library and designed to be integrated on almost every microcontroller, even with very limited resources.

Single instance of µP7proxy library can handle up to 256 independent processors.

⚠ µP7 is capable to work without host. It is possible to save µP7 output to any HW FIFO/cycle buffer and read it later when user will be interested in logs/telemetry.

⚠ It is responsibility of integrator to provide transport between CPU & Hosts! µP7 library has no access to network or any platform specific HW like FIFO/RS-232/etc.!

Memory consumption

Memory consumption can be divided into 3 parts:

1. Internal memory for µP7 (allocated at initialization). The size is always the same – 408 bytes.
2. Internal service memory to describe each module/telemetry/trace item (allocated at initialization) – 40 bytes per instance.
   Next functions are concerned:
   a. uP7TrcRegisterModule
   b. uP7TelCreateCounter
   c. uP7TRC, uP7DBG, uP7INF, uP7WRN, uP7ERR, uP7CRT
   All other µP7 API functions aren't consuming heap or .bss memory.
3. .Bss segment memory consumption
   a. µP7 instance – 264 bytes
   b. One module description – 32 bytes + size of module name (string)
   c. One telemetry description – 24 bytes + size of tel. counter name (string)
   d. One telemetry description – 24 bytes + size of tel. counter name (string)
   e. One trace item description – 16 bytes + 8bytes*(variable arguments count)

⚠ Important notice: all heap memory will be allocated only during uP7initialize() call and won't be allocated later.

Let's make memory consumption simulation using next piece of code:

```
struct stuP7config l_stConfig;
//config initialization
//...
uP7initialize(&l_stConfig);

huP7Module l_hModule1 = NULL;
huP7Module l_hModule2 = NULL;
huP7TelId  l_hCounter = uP7_TELEMETRY_INVALID_ID;

uP7TrcRegisterModule("ModuleA", euP7Level_Trace, &l_hModule1);
uP7TrcRegisterModule("ModuleB", euP7Level_Trace, &l_hModule2);
uP7TelCreateCounter("Counter", 0, 0, 1, 1, true, &l_hCounter1);

for (int i = 0; i < 100; i++)
{
   uP7TRC(0, l_hModule1, "Test message with 2 parameters %d, %d", 10, i);
   uP7TRC(0, l_hModule2, "Test message with 3 parameters %d, %d %s", 10, i, "some
text");

   uP7TelSentSample(l_hCounter, 0.0);
   uP7TelSentSample(l_hCounter, 1.0);
}
```

- uP7initialize call consumes
  - 408 bytes of heap memory
  - 264 bytes in .bss segment
- uP7TrcRegisterModule ("ModuleA") consumes
  - 40 bytes of heap memory
  - 32 bytes + 8 bytes of string (7 chars+0): 40 bytes in .bss segment

- uP7TrcRegisterModule ("ModB") consumes
  - 40 bytes of heap memory
  - 32 bytes + 5 bytes of string (4 chars+0): 37 bytes in .bss segment
- uP7TelCreateCounter consumes
  - 40 bytes of heap memory
  - 24 bytes + 8 bytes of string (7 chars+0): 32 bytes in .bss segment
- uP7TRC (2 var. arguments)
  - 40 bytes of heap memory
  - 16 bytes + 8 * 2: 32 bytes in .bss segment
- uP7TRC (3 var. arguments)
  - 40 bytes of heap memory
  - 16 bytes + 8 * 3: 40 bytes in .bss segment
- uP7TelSentSample – 0 consumption.

In total:

- 608 bytes if heap memory
- 445 bytes if .bss memory

Integration guideline

Integration and *first* use of µP7 may be splitted into several steps:

1. Create µP7 *platform header* file and providing it to µP7 during compilation
2. Integrating µP7 into user firmware project
3. Compiling µP7 preprocessor
4. Using µP7 preprocessor (and user firmware source code as input) generate *session* (YYYYMMDD_HHMMSS.up7) & *description* (uP7preprocessed.h) files
5. Compile µP7 library using *platform header*
6. Compile user firmware using *description* (uP7preprocessed.h) file & µP7 library
7. Integrate µP7 proxy library into user application

Create µP7 platform header

Platform headers are located in folder <uP7>/uP7Lib/Platforms

Platform header is a way to describe your target microcontroller specifics and provide this description to µP7 library.

As example please have a look at <uP7>/uP7Lib/Platforms/x86/uP7platform.h

Example file contains exhaustive comments and it should be really easy to make your own platform header based on this example.

All you need is:

1. Create new subfolder for your platform inside <uP7>/uP7Lib/Platforms/
2. Put into this new subfolder uP7platform.h & CMakeLists.txt files (take as example x86 platform)
3. Modify <uP7>/uP7Lib/Platforms/CMakeLists.txt to include your new platform header into compilation

Integrating µP7 into user firmware project

µP7 library provides flexible way of integration into almost every CPU & OS.

To achieve this library do not use heap, locks, timers … or any other platform/run-time specific primitives, all necessary functionality should be provided by integrator through callbacks and platform header (describer in prev. chapter).

Entry point to µP7 library functionality is API header <uP7>/uP7Lib/Headers/uP7.h

And first what integrator should do is create µP7 instance using function

```
/**
 * \brief P7 initialization function, should be called once at CPU startup
 * @param i_pConfig [in] configuration
 * @return true - uP7 has been initialized, false - failure
*/
bool uP7initialize(const struct stuP7config *i_pConfig);
```

Function will instantiate µP7 as singleton and make library functions visible for entire firmware project.

To initialize µP7 integrator should create & fill structure

```
/*! uP7 configuration structure */
struct stuP7config
{
    /**< Session ID*/
    uint32_t                uSessionId;
    /**< CPU id */
    uint8_t                 bCpuId;
    /**< Context for timer functions */
    void                    *pCtxTimer;
    /**< callback to get system high-resolution timer frequency */
    fnuP7getTimerFrequency  cbTimerFrequency;
    /**< callback to get system high-resolution timer value */
    fnuP7getTimerValue      cbTimerValue;
    /**< Context for memory functions */
    void                    *pCtxMem;
    /**< callback to allocate memory */
    fnuP7malloc             cbMalloc;
    /**< callback to free memory */
    fnuP7free               cbFree;

    /**< ...
    For full list of members please have a look to <uP7>/uP7Lib/Headers/uP7.h
    */
};
```

It consist of mandatory members and optional.

Mandatory:

- Provided by integrator:
  - *bCpuId* – ID of the processor, up to integrator to decide which ID to use
  - *cbTimerFrequency*, *cbTimerValue* – callback to provide to µP7 information about hi-resolution timer to timestamp trace/telemetry/log items.
  - *cbMalloc*, *cbFree* – at uP7initialize() function call µP7 need to allocate few 100x bytes for internal needs.
    cbFree will be called only if integrator decide to call uP7unInitialize() function.
  - *cbSendPacket* – callback will be used by µP7 every time when some data have to be sent to host/HW FIFO.
- Generated by µP7 preprocessor and located in uP7preprocessed.h
  - *uSessionId* (uP7preprocessed.h : g_uSessionId)
  - *pModules, szModules* (uP7preprocessed.h : g_szModules, g_pModules)
  - *pTraces, szTraces* (uP7preprocessed.h : g_szTraces, g_pTraces)
  - *pTelemetry, szTelemetry* (uP7preprocessed.h : g_szTelemetry, g_pTelemetry)

When µP7 will be instantiated – every other function from API list can be called.

⚠ Project example is located at: <uP7>/uP7Example

Compiling µP7 preprocessor

µP7 pre-processor compilation is based on CMake, please refer to Cmake documentation to get build and installing instructions.

There is no specific compiler requirements, the project can be build using even C++ compilers without C++11/14/17/etc. support.

Using µP7 preprocessor

µP7 pre-processor is important part of the project and has to be used every time when firmware micro code has been changed.

The main purposes of µP7 pre-processor are:

- Preprocess user firmware source files to extract information about
  - µP7 Format string
  - µP7 modules names
  - µP7 telemetry counters
- This text information will be used to generate 2 files
  - *session* (YYYYMMDD_HHMMSS.up7) – file later will be used by µP7 proxy for decoding µP7 incoming traffic
  - *description* (uP7preprocessed.h) – will be compiled with firmware micro code providing static information about all trace/log/telemetry calls.
- It allows to
  - Substantially reduce memory footprint
  - Significantly increase performance – no need at run-time analyze format string, allocate memory, making decisions which already have been made by µP7 pre-processor
  - Fundamentally reduce data traffic – only important information is transmitted, no format strings, file names, file lines, module names, telemetry counter names, etc.
  - Apply additional security – without session file it won't be possible to decode µP7 traffic.

Usage of µP7 pre-processor is simple:

```
>uP7preProcessor <config.xml> <sources files dir> <output dir
```

Where

- Config.xml - µP7 pre-processor configuration file, example of such file can be found at <uP7>/uP7Example/uP7preProcessor/uP7Preprocessor.xml
- Source file dir – directory where firmware microcode files are located
- Output dir – output directory, where *session* & *description* files will be saved

 µP7 pre-processor should be launched every time when firmware source files are changed to regenerate *session* & *description* files.

 µP7 pre-processor can be modified by pre-processor to store information about processed filed. It is recommended to include this file to version control system like Git/Svn/etc.

Integrate µP7 proxy library

µP7 proxy library can be integrated in any C++ application, there is no external dependencies, even old C++ compilers can be used.

The main goal for µP7 proxy library is to receive µP7 data from micro-controllers and convert it to appropriate user shape.

1. First of all integrator should link µP7 proxy library with application.
2. Then using API header (<uP7>/uP7ProxyLib/Headers/uP7proxyApi.h) call function

```
/**
    * \brief Create instance of uP7 proxy object.
    * @param i_pP7Args [in] P7 arguments, see P7 documentation for details. May
be NULL
    * @param i_puP7Dir [in] directory where uP7 description files are located,
previously created by uP7preProcessor tool
    * @return new reference counter value
*/
extern "C" P7_EXPORT IuP7proxy* __cdecl uP7createProxy(const tXCHAR  *i_pP7Args,
                                                       const tXCHAR  *i_puP7Dir
                                                      );
```

3. Function will create µP7 proxy object, and using it integrator should register CPU
   ⚠ Please refer to P7 arguments chapter for details.

```
/**
    * \brief Register new CPU and connect it to P7
    * @param i_bCpuId [in] CPU ID
    * @param i_bBigEndian [in] Big-endian CPU flag
    * @param i_qwFreq [in] target CPU clock frequency in Hz. This clock is used
for trace & telemetry timestamp
    * @param i_pName [in] channel name, will be used to display in Baical server
    * @param i_szFifoLen [in] FIFOs size in bytes (IuP7stream  *&o_iStream). Min
value 16384 bytes.
    * @param i_bFifoBiDirectional [in] flag to specify that communication with
CPU is bidirectional and proxy can use
    * it to send control data to CPU like disable telemetry counter, change
verbosity, etc.
    * If fifo is bidirections - set to "true", otherwise - "false"
    * @param o_iFifo [out] FIFO object to be used for CPU communication.
    *                N.B.: Please do not forget to call o_iFifo->Release() right
after UnRegisterCpu();
    * @return true - success, false - error
*/
virtual bool RegisterCpu(uint8_t       i_bCpuId,
                         bool          i_bBigEndian,
                         uint64_t      i_qwFreq,
                         const tXCHAR *i_pName,
                         size_t        i_szFifoLen,
                         bool          i_bFifoBiDirectional,
                         IuP7Fifo     *&o_iFifo
                        );
```

4. Result of the function call will be µP7 FIFO object, and this object should be used to receive & send data from/to µP7 proxy object.

⚠ Example can be found in <uP7>/uP7Example folder

P7 arguments

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:

```
IuP7proxy *iProxy = uP7createProxy(TM("/P7.Verb=0 /P7.Sink=Baical
/P7.Pool=1024"),  TM("."));

iProxy->RegisterCpu(1, false, CpuTimerFrequency(), TM("CPU1"), 0xFFFF, true,
pFifo);
```

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
  - "FileBin" – into a binary file, please use Baical to open it
  - "FileTxt" – into a text file (Windows: UTF-16, Linux: UTF-8)
  - "Console" – into console
  - "Syslog" – into syslog
  - "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=FileTxt” or “/P7.Sink=Console” or “/P7.Sink=Syslog”:

- “/P7.Format” – set log item format for text sink, consists of next sub-elements
    - “%cn” – channel name
    - “%id” – message ID
    - “%ix” – message index
    - “%tf” – full time: YY.MM.DD HH.MM.SS.mils.mics.nans
    - “%tm” – medium time: HH.MM.SS.mils.mics.nans
    - “%ts” – time short MM.SS.mils.mics.nans
    - “%td” – time difference between current and prev. one +SS.mils.mics.nans
    - “%tc” – time stamp in 100 nanoseconds intervals
    - “%lv” – log level
    - “%ti” – thread ID
    - “%tn” – thread name (if it was registered)
    - “%cc” – CPU core index
    - “%mi” – module ID
    - “%mn” – module name
    - “%ff” – file name + path
    - “%fs” – file name
    - “%fl” – file line
    - “%fn” – function name
    - “%ms” – text user message

  Example: “/P7.Format=\"{%cn} [%tf] %lv %ms\"”
- “/P7.Facility” – set Syslog facility, for details: https://tools.ietf.org/html/rfc3164#page-8

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 3 rolling types:
    - Rolling by file size, measured in megabytes (“mb” command postfix). Example:
        - /P7.Roll=100mb”
    - Rolling by logging duration, measured in hours, 1000 hours max (“hr” command postfix). Examples:
        - “/P7.Roll=24hr”

        - "/P7.Roll=1hr"
    - Rolling by exact time measured in hours and minutes ("tm" command postfix), user can specify one or few rolling times. Examples:
        - "/P7.Roll=10:30tm"
        - "/P7.Roll=12:00,00:00tm"
        - "/P7.Roll=00:00,06:00,12:00,18:00tm"
- "/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 0(OFF), max value – 4096. Example: "/P7.Files=4096"
- "/P7.FSize" – option defines maximum P7 logs files cumulative size in MB in destination folder "/P7.Dir" in case if total size of files is larger than specified value - oldest files will be removed. Default value is 0(OFF), max value – 4294967296. This option working only with "/P7.Roll" option. Example: "/P7.FSize=256"

Next parameters are applicable for all trace channels:

- "/P7.Trc.Verb" – verbosity level for all trace channels and associated modules, has next values:
    - "0" – trace
    - "1" – debug
    - "2" – info
    - "3" – warning
    - "4" – error
    - "5" – critical

  Example: "/P7.Trc.Verb=4"

Trace format string specification

µP7 library is capable to send trace/logs messages to host using macros:

- uP7TRC
- uP7DBG
- uP7INF
- uP7WRN
- uP7ERR
- uP7CRT

Messages have variable arguments & format string.

This chapter describes trace/log format specifications.

A format specification, which consists of optional and required fields, has the following form:

```
%[flags][width][.precision][Size modifier]type
```

Each field of the format specification is a character or a number that signifies a particular format option or conversion specifier. The required type character specifies the kind of conversion to be applied to an argument. The optional *flags*, *width*, and *precision* fields control additional format aspects. A basic format specification **contains only** the *percent sign* and a *type character*.

Flags

In a format specification, the first optional field is flags. A flag directive is a character that specifies output justification and output of signs, blanks, leading zeros, decimal points, and octal and hexadecimal prefixes. More than one flag directive may appear in a format specification, and flags can appear in any order.

Flag	Meaning	Default
-	Left align the result within the given field width	Right align
+	Use a sign (+ or –) to prefix the output value if it is of a signed type	Sign appears only for negative signed values (–).
space " "	Use a blank to prefix the output value if it is signed and positive. The blank is ignored if both the blank and + flags appear.	No blank appears.
#	When it's used with the `o`, `x`, or `X` format, the `#` flag uses 0, 0x, or 0X, respectively, to prefix any nonzero output value.	No blank appears.
	When it's used with the `e`, `E`, `f`, `a` or `A` format, the `#` flag forces the output value to contain a decimal point.	Decimal point appears only if digits follow it.
	When it's used with the `g` or `G` format, the `#` flag forces the output value to contain a decimal point and prevents the truncation of trailing zeros.	Decimal point appears only if digits follow it. Trailing zeros are truncated.
	Ignored when used with `c`, `d`, `i`, `u`, or `s`.	
0	If `width` is prefixed by `0`, leading zeros are added until the minimum width is reached. If both `0` and `-` appear, the `0` is ignored. If `0` is specified as an integer format (`i`, `u`, `x`, `X`, `o`, `d`) and a precision specification is also present—for example, `%04.d`—the `0` is ignored.	No padding.

Width

In a format specification, the second optional field is the width specification. The `width` argument is a non-negative decimal integer that controls the minimum number of characters that are output. If the number of characters in the output value is less than the specified width, blanks are added to the left or the right of the values—depending on whether the left alignment flag (-) is specified—until the minimum width is reached. If `width` is prefixed by 0, leading zeros are added to integer or floating-point conversions until the minimum width is reached, except when conversion is to an infinity or NAN.

The width specification never causes a value to be truncated. If the number of characters in the output value is greater than the specified width, or if width is not given, all characters of the value are output, subject to the precision specification.

If the `width` specification is an asterisk (*), an int argument from the argument list supplies the value. The `width` argument must precede the value that's being formatted in the argument list, as shown in this example:

```
printf("%0*d", 2, 3); /* => 03 is output */
```

A missing or small `width` value in a format specification does not cause the truncation of an output value. If the result of a conversion is wider than the `width` value, the field expands to contain the conversion result.

Precision

In a format specification, the third optional field is the `precision` specification. It consists of a period (.) followed by a non-negative decimal integer that, depending on the conversion type, specifies the number of string characters, the number of decimal places, or the number of significant digits to be output.

Unlike the `width` specification, the `precision` specification can cause either truncation of the output value or rounding of a floating-point value. If `precision` is specified as 0 and the value to be converted is 0, the result is no characters output, as shown in this example:

```
printf("%.0d", 0); /* => No characters output */
```

If the `precision` specification is an asterisk (*), an int argument from the argument list supplies the value. In the argument list, the `precision` argument must precede the value that's being formatted, as shown in this example:

```
printf("%.*f", 3, 3.14159265); /* => 3.142 is output */
```

The type determines either the interpretation of precision or the default precision when precision is omitted, as shown in the following table.

Type	Meaning	Default
a,A	The precision specifies the number of digits after the point.	Default precision is 6. If precision is 0, no decimal point is printed unless the # flag is used.
d, i, u, o, x, X, b	The precision specifies the minimum number of digits to be printed. If the number of digits in the argument is less than `precision`, the output value is padded on the left with zeros. The value is not truncated when the number of digits exceeds `precision`.	Default precision is 1.
e, E	The precision specifies the number of digits to be printed after the decimal point. The last printed digit is rounded.	Default precision is 6. If `precision` is 0 or the period

		(.) appears without a number following it, no decimal point is printed.
f	The precision value specifies the number of digits after the decimal point. If a decimal point appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.	Default precision is 6. If `precision` is 0, or if the period (.) appears without a number following it, no decimal point is printed.
g, G	The precision specifies the maximum number of significant digits printed.	Six significant digits are printed, and any trailing zeros are truncated.
s	Not supported yet. ~~The precision specifies the maximum number of characters to be printed. Characters in excess of `precision` are not printed.~~	Characters are printed until a null character is encountered.

Size

In a format specification, the 4th field is an argument size modifier.

The `size` field is optional for some argument types. When no size prefix is specified, the formatter consumes integer arguments—for example, signed or unsigned `char`, `short`, `int`, `long`, and enumeration types—as 32-bit `int` types, and floating-point arguments are consumed as 64-bit `double` types. This matches the default argument promotion rules for variable argument lists.

Some types are different sizes in 32-bit and 64-bit code. For example, `size_t` is 32 bits long in code compiled for x86, and 64 bits in code compiled for x64.

Size prefix	Type specifier	Size in bytes
hh	d,b,i,o,u,x,X	1
h	d,b,i,o,u,x,X	2
	s	1 (ANSI string)
	c	1 (ANSI char)
I32	d,b,i,o,u,x,X	4
l	d,b,i,o,u,x,X	4
	s	Windows: 2 (UTF16) Linux: 4 (UTF32)
	c	Windows: 2 (UTF16) Linux: 4 (UTF32)
ll, I64	d,b,i,o,u,x,X	8
I,z,t	d,b,i,o,u,x,X	X64 System: 8 X32 System: 4
j	d,b,i,o,u,x,X	uintmax_t, intmax_t It is not recommended to use this size prefix due to compilers specifics.

Type

A character that specifies the type of conversion to be applied. The conversion specifiers and their meanings are:

Type character	Argument	Output format
c	character	character

d	Integer	Signed **decimal** integer.
b	Integer	Unsigned **binary** integer. **Warning:** this type isn't standard one!
i	Integer	Signed **decimal** integer.
o	Integer	Unsigned **octal** integer.
u	Integer	Unsigned **decimal** integer.
x	Integer	Unsigned **hexadecimal** integer; uses "abcdef."
X	Integer	Unsigned **hexadecimal** integer; uses "ABCDEF."
s	String	s: char argument is expected (UTF-8) ls, ws: wchar_t argument is expected (UTF-16/UTF-32 – depends on compiler) hs: char argument is expected (ANSI) hs: char argument is expected (ANSI)
e,E	Floating-point	The **double** argument is rounded and converted in the style [-]d.ddde±dd where there is one digit before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero, no decimal-point character appears. An E conversion uses the letter E (rather than e) to introduce the exponent. The exponent always contains at least two digits; if the value is zero, the exponent is 00.
f	Floating-point	The **double** argument is rounded and converted to decimal notation in the style [-]ddd.ddd, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is explicitly zero, no decimal-point character appears. If a decimal point appears, at least one digit appears before it.
g, G	Floating-point	The **double** argument is converted in style f or e (or F or E for G conversions). The precision specifies the number of significant digits. If the precision is missing, 6 digits are given; if the precision is zero, it is treated as 1. Style e is used if the exponent from its conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the fractional part of the result; a decimal point appears only if it is followed by at least one digit.
a, A	Floating-point	For a conversion, the double argument is converted to hexadecimal notation (using the letters abcdef) in the style [-]0xh.hhhhp±; for A conversion the prefix 0X, the letters ABCDEF, and the exponent separator P is used. There is one hexadecimal digit before the decimal point, and the number of digits after it is equal to the precision. The default precision suffices for an exact representation of the value if an exact representation in base 2 exists and otherwise is sufficiently large to distinguish values of type double. The digit before the decimal point is unspecified for nonnormalized numbers, and nonzero but otherwise unspecified for normalized numbers.
p	Pointer type	Displays the argument as an address in hexadecimal digits. The **void *** pointer argument is printed in hexadecimal (as if by %#X or %#lX)

Limitations

Most of the limitations are driven by fact that µP7 pre-processor capable to work only with source code and run-time state isn't accessible, so in your source code you need to take in account next limitations:

- Please do not use function uP7TrcSent directly, instead use macros:
  - uP7TRC
  - uP7DBG
  - uP7INF
  - uP7WRN
  - uP7ERR
  - uP7CRT

  Function uP7TrcSent has no format string, only variable arguments, but µP7 pre-processor need to find and pars format string for further use and macros helping in that.
- Use only static string to specify:
  - format (uP7TRC, uP7DBG, uP7INF, uP7WRN, uP7ERR, uP7CRT)
  - Module name (uP7TrcRegisterModule)
  - Counter name (uP7TelCreateCounter)

```
//Correct:
uP7ERR(0, hModule, "Value %u", 0xFFFFFFFF);

//Incorrect:
char *pFormat = "My Format %u";
uP7ERR(0, hModule, pFormat, 0xFFFFFFFF);
```

- Use only decimal trace ID (uP7TRC, uP7DBG, uP7INF, uP7WRN, uP7ERR, uP7CRT)

```
//Correct:
uP7ERR(0, hModule, "Value %u", 0xFFFFFFFF);

//Incorrect:
uP7ERR(0x0, hModule, "Value %u", 0xFFFFFFFF);
```

- Use only decimal or floating point telemetry values (uP7TelCreateCounter)

```
//Correct:
uP7TelCreateCounter("Counter", 0.0, 0.0, 1.0, 1.0, true, &hCounter1);

//Incorrect:
double dbMin = 0.0;
double dbMax = 1.0;
uP7TelCreateCounter("Counter", dbMin, dbMin, dbMax, dbMax, true, &hCounter1);
```