ULog 文件格式

ULog is the file format used for logging system data.

The format is self-describing, i.e. it contains the format and message types that are logged (note that the system logger allows the *default set* of logged topics to be replaced from an SD card).

It can be used for logging device inputs (sensors, etc.), internal states (cpu load, attitude, etc.) and `printf` log messages.

The format uses Little Endian for all binary types.

数据类型

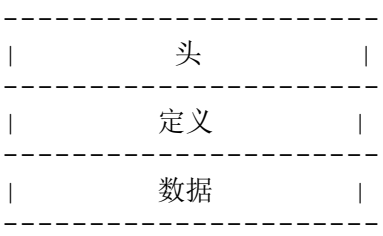
The following binary types are used. They all correspond to the types in C:

类型	大小（以字节为单位）
int8_t, uint8_t	1
int16_t, uint16_t	2
int32_t, uint32_t	4
int64_t, uint64_t	8
float	4
double	8
bool, char	1

Additionally all can be used as an array, eg. `float[5]`. In general all strings (`char[length]`) do not contain a `'\0'` at the end. String comparisons are case sensitive.

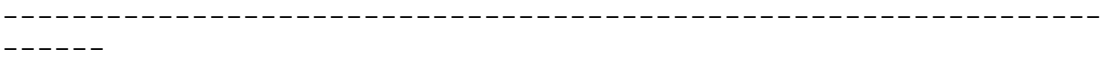
文件结构

The file consists of three sections:



头部分

The header is a fixed-size section and has the following format (16 bytes):



```
| 0x55 0x4c 0x6f 0x67 0x01 0x12 0x35 | 0x01          | uint64_t
|
| File magic(7B)                     | Version (1B) |  Timestamp
(8B) |
-------------------------------------------------------------------
------
```

Version is the file format version, currently 1. Timestamp is a `uint64_t` integer, denotes the start of the logging in microseconds.

定义部分

Variable length section, contains version information, format definitions, and (initial) parameter values.

The Definitions and Data sections consist of a stream of messages. Each starts with this header:

```
struct message_header_s {
  uint16_t msg_size;
  uint8_t msg_type
};
```

`msg_size` is the size of the message in bytes without the header (`hdr_size`= 3 bytes). `msg_type` defines the content and is one of the following:

- 'B'：标记 bitset 报文。
- `struct ulog_message_flag_bits_s {`
- `  uint8_t compat_flags[8];`
- `  uint8_t incompat_flags[8];`
- `  uint64_t appended_offsets[3]; ///< file offset(s) for appended data if appending bit is set`
- `};`

这条消息**必须**是头后面的第一条消息，这样才有固定的常数偏移量。

- `compat_flags`: 兼容的标志位。 它们目前都没有定义，都必须设置为 0。 这些位可用于将来的 Ulog 更改，即与现有解析器兼容。 这意味着, 如果设置了一个未知位，解析器就可以忽略。
- `incompat_flags`: 不兼容的标志位。 如果日志包含附加数据，并且至少有一个 `appended_offset` 是非零的，那么索引 0 的 LSB 位被设置为 1。 其他位都是未定义的，必须将设置为 0。 如果解析器发现这些位置 1，它必须拒绝解析日志。 这可用于引入现有解析器无法处理的重大更改。
- `appended_offsets`: 附加数据的文件偏移量 (基于 0)。 如果没有附加数据，则所有偏移量必须为零。 这可以用于消息中途暂停的情况下可靠的添加数据。

  附加数据的过程应该做到：

  - 置位相关的 `incompat_flags` 位，
  - 设置 `append_offsets` 的第一个元素为日志文件相对于 0 的长度，
  - 然后为数据部分添加有效的任何类型的消息。

这使得在将来的 Ulog 规范中在末尾添加更多的字段成为可能。 这意味着解析器必须不能假定此消息的长度是固定的。 如果消息比预期的长（当前为 40 字节），则必须忽略超过的字节。

- 'F': 可以在另一个定义中作为嵌套类型记录或使用的单个 (组合) 类型的格式定义。
- `struct message_format_s {`
- `  struct message_header_s header;`
- `  char format[header.msg_size];`
- `};`

`format`: 具有以下格式的纯文本字符串：`message_named: field0;field1;` 可以有任意数量的字段 (至少 1 个)，采用 `;` 分隔。 字段的格式为：`type field_name` 或者 `type[array_length] field_name` 数组（只支持固定大小的数组）。 `type` 是一种基本的二进制类型或者是 `message_name` 的其他类型定义（嵌套使用）。 一个类型可以在定义之前使用。 可以任意嵌套，但没有循环依赖。

有些字段名是特殊的：

- `timestamp`：每个消息报文 (`message_add_logged_s`) 必须包含时间戳字段 (不必是第一个字段)。 它的类型可以是：`uint64_t` (目前唯一使用的)，`uint32_t`, `uint16_t` 或者是 `uint8_t` 。 它的单位一直是微秒，除了 `uint8_t`，它的单位是毫秒。 日志写入器必须确保足够频繁的写入报文使其能够检测到绕回，并且日志的读取器必须能够处理绕回 (还要把丢帧考虑在内)。 对于具有相同 `msg_id` 报文的时间戳必须是单调递增的。
- Padding：以 `_padding` 开始的字段名应该不被显示并且必须被读取器忽略。 写入器可以通过插入这个字段确保正确对齐。

  如果 Padding 字段是最后一个字段，则不会记录该字段，以避免写入不必要的数据。 这意味着 `message_data_s.data` 会因为填充大小而更短。 但是当报文在嵌套定义中使用时任然需要填充。

- 'I'：信息报文。
- `struct message_info_s {`
- `  struct message_header_s header;`
- `  uint8_t key_len;`
- `  char key[key_len];`
- `  char value[header.msg_size-1-key_len]`
- `};`

  `key` 是纯字符串，就像报文格式 (也可以是第三方类型)，但只包含一个没有结束符 `;` 的字段。 `float[3] myvalues.value` 包含 `key` 所描述的字段

  需要注意的是包含特定 key 的报文信息在整个日志中最多只能出现一次。 解析器可以将报文信息存储为字典。

  预定义的信息报文有：

键	描述	示例值
char[value_len] sys_name	系统名称	"PX4"

键	描述	示例值
char[value_len] ver_hw	硬件版本 (主板)	"PX4FMU_V4"
char[value_len] ver_hw_subtype	主办子版本 (变化的)	"V2"
char[value_len] ver_sw	软件版本 (git 标签)	"7f65e01"
char[value_len] ver_sw_branch	git branch	"master"
uint32_t ver_sw_release	软件版本 (见下文)	0x010401ff
char[value_len] sys_os_name	操作系统名称	"Linux"
char[value_len] sys_os_ver	操作系统版本 (git 标签)	"9f82919"
uint32_t ver_os_release	操作系统版本 (见下文)	0x010401ff
char[value_len] sys_toolchain	工具链名称	"GNU GCC"
char[value_len] sys_toolchain_ver	工具链版本	"6.2.1"
char[value_len] sys_mcu	芯片名称和修订	"STM32F42x, rev A"
char[value_len] sys_uuid	车辆的唯一标识符 (例如 MCU ID)	"392a93e32fa3"...
char[value_len] log_type	Type of the log (full log if not specified)	"mission"
char[value_len] replay	File name of replayed log if in replay mode	"log001.ulg"
int32_t time_ref_utc	UTC Time offset in seconds	-3600

`ver_sw_release` 和 `ver_os_release` 的类型是：0xAABBCCTT，其中 AA 是主要的，BB 是次要的，CC 是补丁，TT 是类型。
类型定义如下：`>= 0`：development 版本，`>= 64`：alpha 版本，`>= 128`：beta 版本，`>= 192`：RC 版本，`== 255`：release 版本。
So for example 0x010402ff translates into the release version v1.4.2.

This message can also be used in the Data section (this is however the preferred section).

- 'M'：多报文信息。
- ```
  struct ulog_message_info_multiple_header_s {
    struct message_header_s header;
    uint8_t is_continued; ///< can be used for arrays
    uint8_t key_len;
    char key[key_len];
    char value[header.msg_size-2-key_len]
  };
  ```

  与报文消息相同，不同的是可以有多个具有相同密钥的消息 (解析器将它们存储为列表) 。 `is_continued` 可以用于分割报文：如果置 1，则它是具有相同键的前一条报文的一部分。 解析器可以将所有多报文信息存储为一个 2D 列表，使用与日志中报文相同的顺序。

- 'P'：报文参数。 格式与 `message_info_s` 相同。 如果参数在运行时动态变化，则此报文也可用于 Data 部分。 数据类型限制为：`int32_t`，`float` 。

This section ends before the start of the first `message_add_logged_s` or `message_logging_s` message, whichever comes first.
  ```

数据部分

The following messages belong to this section:

- 'A'：按名称订阅消息，并给它一个在 message_data_s 中使用的 id。 这必须在第一个对应的 message_data_s 之前。
- struct message_add_logged_s {
- struct message_header_s header;
- uint8_t multi_id;
- uint16_t msg_id;
- char message_name[header.msg_size-3];
- };

  multi_id：相同的消息格式可以有多个实例，例如系统有两个相同类型的传感器。 默认值以及第一个实例一定是 0. msg_id：匹配 message_data_s 数据的惟一 id。 第一次使用一定要设置为 0，然后递增。 相同的 msg_id 不能用于两次不同的订阅，甚至在取消订阅后也不行。 msg_name：订阅的消息名称。 必须匹配其中一个 message_format_s 的定义。

- 'R'：取消订阅一条消息，以标记它将不再被记录 (当前未使用)。
- struct message_remove_logged_s {
- struct message_header_s header;
- uint16_t msg_id;
- };
- 'D'：包含日志数据。
- struct message_data_s {
- struct message_header_s header;
- uint16_t msg_id;
- uint8_t data[header.msg_size-2];
- };

msg_id：由 message_add_logged_s 报文定义。 data 包含由 message_format_s 定义的二进制日志消息。 有关填充字段的特殊处理，请参见上文。

- 'L'：字符串日志报文，比如打印输出。
- struct message_logging_s {
- struct message_header_s header;
- uint8_t log_level;
- uint64_t timestamp;
- char message[header.msg_size-9]
- };

timestamp: 以微秒为单位，log_level: 和 Linux 内核一样。

名称	对应值	含义
EMERG	'0'	系统无法使用
ALERT	'1'	操作必须立即执行
CRIT	'2'	紧急情况
ERR	'3'	错误情况

名称	对应值	含义
WARNING	'4'	警告情况
NOTICE	'5'	正常但重要的情况
INFO	'6'	信息
DEBUG	'7'	调试级别的消息

- 'S'：报文同步，使读取器可以通过搜索下一个同步报文 (目前没有使用) 从损坏的报文中恢复。
- `struct message_sync_s {`
- `  struct message_header_s header;`
- `  uint8_t sync_magic[8];`
- `};`

`sync_magic`：未定义。

- 'O'：在给定毫秒的时间内对丢包（日志报文丢失）的标记。 例如当设备不够快的情况下会出现丢包。
- `struct message_dropout_s {`
- `  struct message_header_s header;`
- `  uint16_t duration;`
- `};`

- 'I'：信息报文。 见上文。
- 'M'：多报文信息。 见上文。
- 'P'：报文参数。 见上文。

解析器的要求

A valid ULog parser must fulfill the following requirements:

- 必须忽略未知消息 (但可以打印警告) 。
- 解析未来/未知的文件格式版本 (但可以打印警告) 。
- 必须拒绝解析包含未知不兼容位集 (`ulog_message_flag_bits_s` 报文中的 `incompat_flags`) 的日志，这意味着日志包含解析器无法处理的突发改变。
- 解析器必须能够正确处理报文突然结束的日志。 未完成的报文应该丢弃。
- 对于附加数据:解析器可以假设数据部分存在，即在定义部分之后的位置有一个偏移点。

  必须将附加数据视为常规数据部分的一部分。

已知的实现

- PX4 Firmware: C++
  - 日志模块
  - 回放模块
  - hardfault_log module：添加硬故障崩溃的数据
- pyulog：Python，使用 CLI 脚本的 Ulog 解析库。
- FlightPlot: Java，日志绘图仪。
- pyFlightAnalysis：Python，日志绘图仪和基于 pyulog 的三维可视化工具。

- MAVLink：通过 MAVLink 进行 ULog 流的消息 (注意，不支持追加数据，至少不支持截断消息)。
- QGroundControl：C++，通过 MAVLink 的 Ulog 流和最小的 GeoTagging。
- mavlink-router：C++，通过 MAVLink 的 ULog 流。
- MAVGAnalysis：Java，通过 MAVLink 的数据流和日志的绘制、分析。
- PlotJuggler: 绘制日志和时间序列的 C++/Qt 应用。 自版本 2.1.3 支持 ULog。

文件格式版本历史

版本 2 中的改变

Addition of `ulog_message_info_multiple_header_s` and `ulog_message_flag_bits_s` messages and the ability to append data to a log. This is used to add crash data to an existing log. If data is appended to a log that is cut in the middle of a message, it cannot be parsed with version 1 parsers. Other than that forward and backward compatibility is given if parsers ignore unknown messages.