# EasyIoT SDK 说明

本 SDK 使用 ANSI C 实现了 EasyIoT 终端接口协议，使用简单的 API 抽象，实现了NB-IoT数据上送与下行指令处理、响应等。核心文件 easyiot.c 代码行数仅千行左右，结构精简，易于阅读与修改。

终端侧代码，请包含头文件 easyiot.h，所有需要导出与使用的API均在此文件中有声明。在EasyIoT平台产品定义处，取出导出头文件中的宏定义部分，如本SDK中的example.h 。

本SDK代码在gcc、msvc、armcc中编译测试通过。

**开源声明**：EasyIoT终端接口协议与本SDK软件实现，由中国电信股份有限广东研究院完成，并按Apache License 2.0开源协议发布，未经单独授权，不得以盈利为目的单独销售该SDK软件。下载本SDK软件，表明您已经默认接受上述声明。

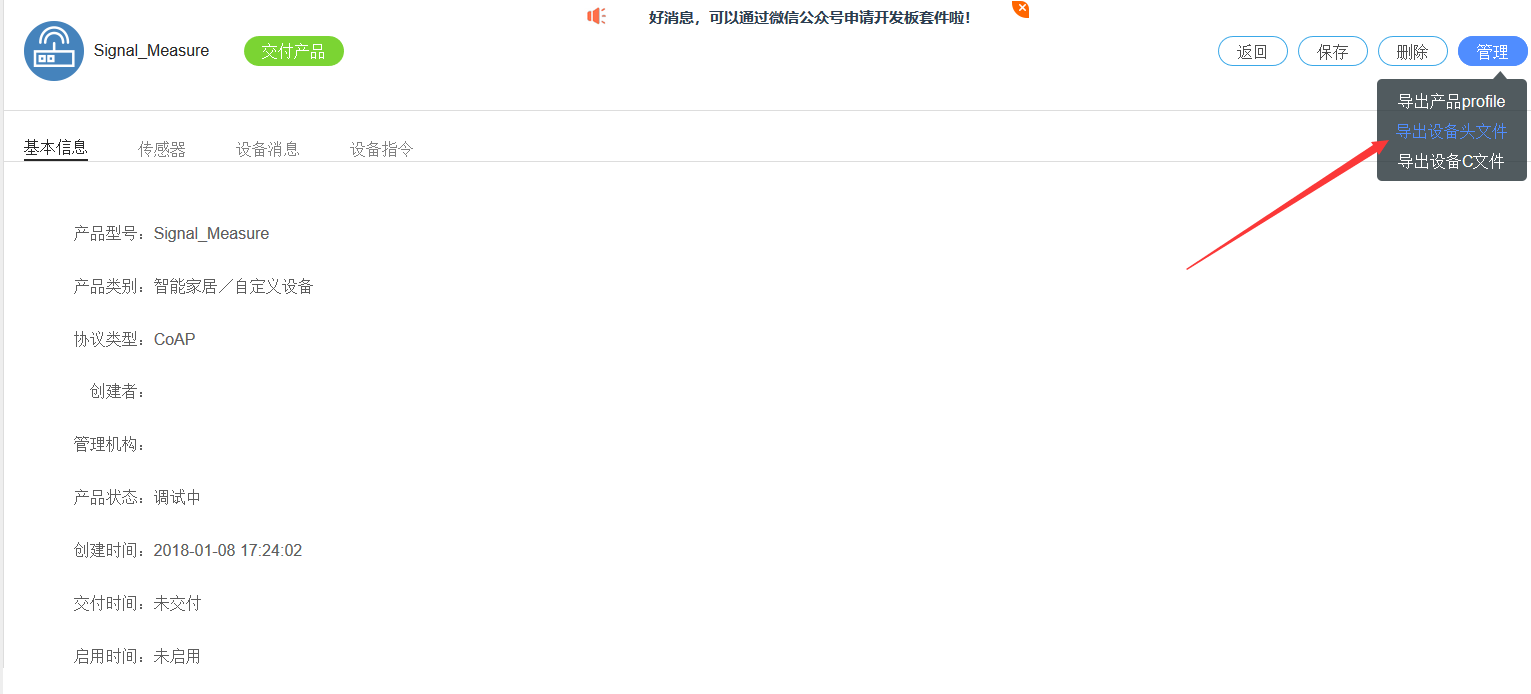
# 基础用法

## 平台侧

按EasyIoT 文档中心所示，明确需求，编辑产品后，即可开始终端侧的开发。



在编辑完产品后，在平台的产品管理中，导出设备头文件，设备头文件中包含了设备上行消息与下行指令的指令标识宏定义导出，如下图所示：



## 初始化

|  |
| --- |
| // 初始化，以下所有初始化都是全局操作，EasyIotInit 以 IMEI和 IMSI进行全局初始化  EasyIotInit("863703035110241", "460111174809981");  // 设置上传信息获取回调函数 以下三个回调函数可选，若不设置，则 时间戳、电量、信号强度取 0值。  setsTimestampCb(getTimestamp); setSignalCb(getSignal); setBatteryCb(getBattery);  // 设置日志输出，设置指令输出，在MCU中需要分开  setLogSerialOutputCb(outputfunc); setNbSerialOutputCb(outputfunc);    // 设置ACK数据确认回调函数，若无需处理ACK，则可选  setAckHandler(ack\_handler);  // 设置指令回调函数，若无指令，则可选，指令的回调ID，取 EasyIoT 平台中指令的ID。 setCmdHandler(CMD\_CMDID, cmd\_handler); |

## 数据上报

|  |
| --- |
| void data\_report\_static(uint8\_t \*buf, uint16\_t inMaxLength)  {  struct Messages\* msg;    msg = NewMessageStatic(buf, inMaxLength);    // 设置上送数据ID，CMT\_USER\_UP设置为用户数据上报，为固定值；MSG\_MSGID为导出头文件中对应消息的宏定义，需要修改。  setMessages(msg, CMT\_USER\_UP, MSG\_MSGID);  // 传感器数据上送，LON\_TLV\_PARAMID为平台导出头文件中对应定义的宏  AddFloat(msg, LON\_TLV\_PARAMID, 123.123f);  AddFloat(msg, LAT\_TLV\_PARAMID, 321.456f);  AddInt8(msg, STATUS\_TLV\_PARAMID, 129);  AddFloat(msg, 4, 321.456f);  // 添加一个字符串，传感器ID为固定数字5；  AddString(msg, 5, "HELLO,WORLD");  // 将msg对象的值，上送到平台。  pushMessages(msg);    // 可选，函数返回时，buf空间被收回，则msg自动被free  FreeMessage(msg);  } |

## 指令处理

|  |
| --- |
| void show\_cmd\_req\_info(struct Messages\* req)  {  int ret; int8\_t vint8;    vint8 = 0;  // 以指定的type值，获取一个 Int8 的TLV值  ret = GetInt8(req, STATUS\_TLV\_PARAMID, &vint8);  if (ret < 0) {  printf("fetch int8 %d faied.\n", STATUS\_TLV\_PARAMID);  } else {  printf("fetch %d value %d\n", STATUS\_TLV\_PARAMID, vint8);  }  } |

## 内存碎片处理

终端与EasyIoT平台的通讯，使用了TLV，即Type-length-value数据报，在TLV的处理过程中使用了malloc进行动态内存分配，长时间运行有可能产生内存碎片，导致malloc分配失败。对此，使用SDK的static系列API，即可规避此问题。

static系列api，使用预分配栈空间或全局内存空间进行TLV的管理，程序编译通过后，其运行时内存布局即为确定布局，无内存碎片问题。

# API

## 结构体

enum CoapMessageType { CMT\_USER\_UP = 0xF0,

CMT\_USER\_UP\_ACK = 0xF1,

CMT\_USER\_CMD\_REQ = 0xF2,

CMT\_USER\_CMD\_RSP = 0xF3,

CMT\_SYS\_CONF\_REQ = 0xF4,

CMT\_SYS\_CONF\_RSP = 0xF5,

CMT\_SYS\_QUERY\_REQ = 0xF6,

CMT\_SYS\_QUERY\_RSP = 0xF7

};

CoAP数据类型，分别为用户数据上行，平台下发数据收到确认ACK，用户自平台下发到设备的指令，用户对指令的响应，CMT\_SYS的则是EasyIoT公有数据格式，可选择性实现，具体定义在文档中心-终端接口协议中有描述。

struct TLV {

uint16\_t length;

uint8\_t type;

uint8\_t vformat;

uint8\_t \*value;

};

TLV结构体，EasyIoT平台与终端通讯格式，使用了TLV，其中vformat字段标注了本TLV中存储了何种格式的数据，其定义在枚举 enum TlvValueType中，包括了如 INT、FLOAT、DOUBLE等常见格式。

|  |
| --- |
| #define MESSAGE\_MAX\_TLV 32 struct Messages {  // dtag\_mid，在上行数据中使用 dtag 语义，用于在一个较短的时间内，去除重复上行的数据  // 在下行指令中，取其mid语义，上行的指令响应必须使用同样的mid值，用以关联指令的执行结果。  uint16\_t dtag\_mid;    // msgid，消息ID，即为整个消息的Type，在EasyIoT平台定义消息时，此值会被定义  // 在导出的头文件中，亦有此值的宏定义，代码中建议使用宏定义。 uint8\_t msgid;    // 当前Message对象中的TLV个数总和 uint8\_t tlv\_count;    /\* 静态内存分配区，在 MessageMalloc函数中，实现了一个简单的静态内存分配机制，即只分配，不释放。 \*/  // 内存buffer起始地址，一旦赋值，将不再变动，具体内存使用，由 sbuf\_offset 值确定  uint8\_t\* sbuf; uint16\_t sbuf\_offset;  // 原始 sbuf 的最大长度  uint16\_t sbuf\_maxlength;  // 是否使用静态内存分配机制  uint8\_t sbuf\_use; /\* 静态内存分配区 \*/    // enum TlvValueType 值，用于区分是何种类型数据 uint8\_t msgType;  // TLV对象指针数组  struct TLV\* tlvs[MESSAGE\_MAX\_TLV]; }; |

Message结构体为EasyIoT平台数据格式抽象，各字段释义见注释；

enum LoggingLevel { LOG\_TRACE,

LOG\_DEBUG,

LOG\_INFO,

LOG\_WARNING,

LOG\_ERROR,

LOG\_FATAL };

日志输出等级，可使用SetLogLevel控制日志输出等级。

## 函数

void EasyIotInit(const char\* imei, const char\* imsi);

SDK初始化，使用IMEI与IMSI初始化，

|  |
| --- |
| typedef uint64\_t(\*TimestampCbFuncPtr)(void); typedef uint8\_t(\*BatteryCbFuncPtr)(void); typedef int32\_t(\*SignalCbFuncPtr)(void);  typedef void(\*OutputFuncPtr)(const uint8\_t\* buf, uint16\_t length); typedef void(\*CmdHandlerFuncPtr)(struct Messages\* req);    void setsTimestampCb(TimestampCbFuncPtr func); void setSignalCb(SignalCbFuncPtr func); void setBatteryCb(BatteryCbFuncPtr func); void setNbSerialOutputCb(OutputFuncPtr func); void setAckHandler(CmdHandlerFuncPtr func); int setCmdHandler(int cmdid, CmdHandlerFuncPtr func); |

setsTimestampCb，设置时间戳回调函数，请返回当前系统毫秒数，若系统只支持到秒数，返回值乘以1000转换到毫秒数即可。

setSignalCb，设置信号强度回调函数，请返回RSRP值 setBatteryCb，设置电池电量回调函数，返回0~100之间的数值，代表电量百分比 setNbSerialOutputCb，设置与NB通信的输出函数，设置日志输出回调函数

setAckHandler，设置ACK处理回调函数 setCmdHandler，设置指令处理回调函数

struct Messages\* NewMessage(void);

struct Messages\* NewMessageStatic(uint8\_t\* buf, uint16\_t inMaxLength); void FreeMessage(struct Messages\* msg);

void setMessages(struct Messages\* msg, enum CoapMessageType type, uint8\_t msgid);

新建Message对象，Message对象是EasyIoT中数据的封装，上行数据、下行指令都使用此结构体；后者在指定的内存空间内初始化，并返回内存空间起始值；一个Message对象最多容纳32各 TLV对象，在头文件中修改宏定义即可去除此限制；但嵌入式环境中请留意内存的使用。

FreeMessage释放Message对象所占用的内存空间，但若对应的Message对象使用静态内存分配，即使用NewMessageStatic构造而得，则本函数仅将初始内存段置0。 setMessages设置Message对象类型，使用枚举值CoapMessageType，具体类型释义请参考文档中心的终端接口协议。 设置Message对象ID，请使用平台导出文件中的指定值。

int AddInt8(struct Messages\* msg, uint8\_t type, int8\_t v);

int AddInt16(struct Messages\* msg, uint8\_t type, int16\_t v);

int AddInt32(struct Messages\* msg, uint8\_t type, int32\_t v);

int AddFloat(struct Messages\* msg, uint8\_t type, float v);

int AddDouble(struct Messages\* msg, uint8\_t type, double v);

int AddString(struct Messages\* msg, uint8\_t type, const char\* v);

int AddBinary(struct Messages\* msg, uint8\_t type, const char\* v, uint16\_t length);

将一个传感器值，添加到指定的Message对象中

int GetInt8(const struct Messages\* msg, uint8\_t type, int8\_t\* v);

int GetInt16(const struct Messages\* msg, uint8\_t type, int16\_t\* v);

int GetInt32(const struct Messages\* msg, uint8\_t type, int32\_t\* v);

int GetLong64(const struct Messages\* msg, uint8\_t type, int64\_t\* v);

int GetFloat(const struct Messages\* msg, uint8\_t type, float\* v);

int GetDouble(const struct Messages\* msg, uint8\_t type, double\* v);

int GetString(const struct Messages\* msg, uint8\_t type, char\*\* v);

int GetBinary(const struct Messages\* msg, uint8\_t type, uint8\_t\*\* v);

从一个Message对象中，获取指定传感器的值，以指定的类型。

int pushMessages(struct Messages \*msg);

pushMessages，据上送函数，上送EasyIoT 平台定义的 Messages，最终内部会调用 CoapOutput 输出指令到硬件。

int a2b\_hex(const char\* s, char\* out, int inMaxLength);

a2b\_hex，将每两字节ASCII表示的HEX数据，转换到1字节的二进制内存byte数据，如2字节的字符串”AA”，将转换到1字节的byte数据 0xAA；

int MessagesSerialize(const struct Messages\* msg, char\* inBuf, uint16\_t i nMaxLength);

int MessagesDeserialize(const char\* inBuf, uint16\_t inLength, struct Mess ages\* out);

MessagesSerialize，数据序列化，将Messages对象序列化到内存中，使之可被网络传输； MessagesDeserialize，数据反序列化，将从平台得到的数据，使用a2b\_hex转换为二进制内存数据后，使用此函数反序列化为Message对象。

int CoapInput(struct Messages\* msg, uint8\_t \*data, uint16\_t inLength); int CoapHexInput(const char\* data);

int CoapHexInputStatic(const char\* data, uint8\_t\* inBuf, uint16\_t inMaxLe ngth);

CoapInput，CoapHexInput，CoapHexInputStatic，将平台下行的CoAP数据，输入到EasyIoT SDK中，本函数首先对数据进行反序列化操作，得到对应的Message对象，再根据Message对象的类型，执行具体的操作，如调用ACK回调函数，调用下行指令回调函数。

void SetLogLevel(enum LoggingLevel level);

int Logging(enum LoggingLevel level, const char\* fmt, ...); void setLogSerialOutputCb(OutputFuncPtr func);

日志处理，使用 SetLogLevel 函数控制日志输出级别，使用 Logging 函数输出日志。

int CoapOutput(uint8\_t \*inBuf, uint16\_t inLength);

底层接口 输出CoAP 数据流，不同的模组，需要使用不同的PORTING