# 汽车电子及C语言知识文档

目录

[汽车电子及C语言知识文档 1](#_Toc16197)

[1. 业务大概 4](#_Toc9940)

[2. 汽车电子 4](#_Toc5779)

[2.1. UDS 4](#_Toc5757)

[2.2. 刷写过程中 4](#_Toc8700)

[2.3. Bootloader 刷写怎么实现 4](#_Toc13240)

[2.4. CAPL：Bootloader上位机刷写 5](#_Toc8199)

[2.5. 段 5](#_Toc31747)

[2.5.1. .data段 5](#_Toc13576)

[2.5.2. .bss 5](#_Toc21506)

[2.6. AUTOSAR架构 5](#_Toc17958)

[2.6.1. AUTOSAR架构怎么跳转？ 5](#_Toc15089)

[2.7. CAN 7](#_Toc4510)

[2.7.1. CAN的特点 7](#_Toc27228)

[2.7.2. 错误 8](#_Toc5633)

[2.7.3. CAN和CANFD的区别 9](#_Toc19682)

[2.7.4. 当CAN总线没有报文时 9](#_Toc10369)

[2.7.5. BUSOFF 10](#_Toc19410)

[2.7.6. 造成原因 10](#_Toc23176)

[2.7.7. 如何恢复 10](#_Toc13019)

[2.7.8. 10](#_Toc4400)

[2.8. 信息安全 10](#_Toc8755)

[2.8.1. AES 10](#_Toc1367)

[2.8.2. 分类 10](#_Toc2769)

[2.9. ISO15765 11](#_Toc22877)

[2.9.1. 网络层协议 11](#_Toc19898)

[2.9.2. 应用层时间参数 13](#_Toc6611)

[2.9.3. 会话层时间参数 14](#_Toc7788)

[2.10. 15](#_Toc26786)

[2.11. 15](#_Toc2541)

[3. C语言 15](#_Toc20086)

[3.1. 指针与其他 15](#_Toc20235)

[3.1.1. 指针与常量 15](#_Toc19568)

[3.1.2. 指针与数组 15](#_Toc22982)

[3.1.3. 什么是野指针？ 16](#_Toc22631)

[3.1.4. 16](#_Toc776)

[3.1.5. 堆栈溢出 16](#_Toc72)

[3.2. 自增 16](#_Toc32529)

[3.2.1. 自增的副作用 16](#_Toc5454)

[3.3. 函数 17](#_Toc30781)

[3.3.1. Malloc/Calloc 17](#_Toc1581)

[3.3.2. Strlen 17](#_Toc277)

[3.4. 大端/小端 18](#_Toc1894)

[3.4.1. 大端字节序 18](#_Toc8135)

[3.4.2. 小端字节序 18](#_Toc25433)

[3.4.3. 18](#_Toc1654)

[3.4.4. 18](#_Toc22484)

[3.5. 堆栈 18](#_Toc26377)

[3.5.1. 栈(Stack) 18](#_Toc21807)

[3.5.2. 堆(Heap) 19](#_Toc1626)

[3.5.3. 19](#_Toc21303)

[3.6. 19](#_Toc4596)

[3.7. 19](#_Toc21105)

[3.8. 19](#_Toc5323)

[4. Linux 19](#_Toc5248)

[4.1.1. TCP/UDP 19](#_Toc2694)

[4.1.1.1. TCP 19](#_Toc11443)

[4.1.1.2. UDP 20](#_Toc20061)

[4.1.1.4. UDP 21](#_Toc20996)

[4.1.5. 21](#_Toc18833)

[5. 21](#_Toc26127)

# 业务大概

* 诊断
* Bootloader刷写

# 汽车电子

* 配置普华工具链中的DCM模块。
* DCM：寻址方式、安全等级、诊断服务、BufferSize、DID、22安全等级

## UDS

* 27服务：信息安全 27奇数为请求种子，偶数为发送key
* 22服务：后面跟DID
* 2E服务：手写写入DID

怎么实现：操作指针写入到相关地址,没有NVM。

* 34服务：34 00:标识符 0x44：内存地址-内存大小

栈溢出：发送34服务，擦内存时进了trap，发现是中断栈溢出。原因是DCM\_Callout 中的0X34擦函数在擦的时候中断栈的大小配小了。CANTP能进 。

* 36服务：发送数据

下载：每次请求，规定Block大小将数据下载到规定的地址。

* 37服务：结束下载。
* 31服务：依赖性检查中，RSA3072的值传过来，静态代码中是uint8类型，数据溢出，排查到地址中数据被存放，于是最后排查到静态代码。

## 刷写过程中

* 基于HSM做完整性检查和依赖性检查
* 完整性检查：对整个文件进行检查
* 依赖性检查：对硬件和文件进行检查。RSA签名；防止你刷错文件,确认硬件，软件一致

## Bootloader 刷写怎么实现

* 0x10 ：切换到编程会话模式。
* 0x85：控制DTC
* 0x28：控制通信
* 0x27：提供安全认证（如 Seed & Key 验证）。
* 0x31 ：擦除存储区域或激活新固件。
* 0x34 ：准备接收新的固件数据。
* 0x36 ：按块传输新固件数据。
* 0x37 ：通知 ECU 数据传输完成并校验数据完整性。
* 0x31 ：完整性依赖性检查
* 在boot中，先走完整性检查，再走依赖性检查

## CAPL：Bootloader上位机刷写

## 段

### .data段

* 存放已初始化的全局变量和静态变量

### .bss

* 存放未初始化的全局变量和静态变量

## AUTOSAR架构

### AUTOSAR架构怎么跳转？

#### 应用层

* 应用层负责生成并处理高层业务逻辑的数据。应用层将数据（消息）传递给\*\*RTE（Runtime Environment）\*\*模块，RTE模块会负责与其他层之间的交互。

#### RTE

* RTE是一个桥梁，负责应用层和其他模块（例如网络层、传输层、通信模块等）之间的数据传输。RTE接收来自应用层的消息，并将其转发到合适的模块。它提供了抽象层，隐藏了底层硬件和协议的细节。

· **发送消息**：应用层通过RTE调用相应的通信接口，传输消息到网络层或其他模块。

· **接收消息**：当下层传输完成消息后，RTE会将接收到的消息传递给上层的应用模块。

#### COM

* COM模块负责管理通信会话和数据交换。它接收到来自RTE的消息后，决定将消息传递给哪一层，并且根据不同的协议（如CAN、Ethernet等）进行处理。
* **消息封装**：COM模块会对应用层的消息进行封装，转换成适合底层协议传输的格式（例如：对数据进行分段或加上必要的头部信息）。

#### PDUR

* PDUR模块（PDU Router）负责将COM模块处理好的消息进一步路由到合适的传输协议层。PDU指的是协议数据单元（Protocol Data Unit），它是消息传输的基本单元。PDUR会根据配置的路由规则，将PDU传递到正确的协议模块。

#### CANTP

* 如果是CAN协议，**CANTP**（CAN Transport Protocol）模块负责对大的数据块进行分段或重组。它会将大的PDU拆分成多个较小的CAN数据帧，或者在接收时，将多个CAN数据帧重新组合成完整的数据包。
* 用于在 CAN 总线上传输超过 8 字节的数据。

#### CANIF

* **CANIF**与底层的CAN硬件进行交互，最终将消息发送到目标ECU或设备。硬件控制器会根据目标CAN ID，将消息发送到正确的目标节点。

#### 总结

* 应用层–>RTE–>COM–>PDUR–>CANTP–>CAN Interface–>CAN Driver–>CAN controller–>CAN transceiver–>CAN BUS Line
* DCM：负责管理诊断通信。
* DEM：DEM 负责管理 ECU 内部所有的诊断事件,主要是DTC管理。

## CAN

### CAN的特点

#### 多主控制

* 在总线空闲时，所有的单元都可开始发送消息（多主控制）；最先访问总线的单元可获得发送权；多个单元同时发送时，发送高优先级ID消息的单元可获得发送权。

#### 消息发送

* 所有消息以固定格式发送，总线空闲时，所有与总线相连的单元都可以发送新消息。同时发送消息时，根据标识符决定优先级；对各个消息ID的每个位进行逐个仲裁比较。仲裁获胜的单元继续发送消息，失利的单元停止发送进行接收工作。

#### 系统的柔软性

* 与总线相连的单元没有类似于“地址”的信息，因此在总线上增加单元时，连接在总线上的其他单元的软硬件及应用层都不需要改变。

#### 通信速度

* 根据整个网络的规模，可设定适合的通信速度。所有单元必须设定成统一的通信速度。即使有一个单元的通信速度与其他的不一样，此单元也会输出错误信号。不同网络间则有不同的通信速度。

#### 远程数据请求

* 可通过发送“遥控帧”请求其他单元发送数据。

#### 错误检测/错误通知/错误恢复

* 所有单元都可以检测错误。
* 检测出错误的单元会立即通知其他单元。
* 正在发送的单元一旦检测出错误，会强制结束当前的发送。强制结束的单元会不断的反复重新发送此消息直到成功发送为止。

#### 故障封闭

* CAN可以判断出错误类型是总线上暂时的数据错误还是持续的数据错误。
* 当发生持续错误时，将引起此故障的单元从总线上隔离出去。

#### 连接

* CAN总线可以同时连接多个单元的总线。可连接的单元总数理论上是没有限制的。但实际上可连接的单元数受到总线上的时间延迟及电气负载的限制。
* 降低通信速度，可连接的单元增加，提高则减少。

### 错误

#### 主动错误状态

* 主动错误状态可以正常参加总线通信的状态
* 发送0-127且接收 0-127

#### 被动错误状态

* 被动错误状态是易引起错误的状态。
* 处于被动错误状态的单元虽然能参加总线通信，但为不妨碍其他单元通信，接收时不能积极地发送错误通知。
* 即使检测出错误，而其他处于主动错误状态的单元如果没发现错误，整个总线也被认为没有错误。
* 发送128-255或接收128-255

#### 总线关闭状态

* 不能参加总线上通信的状态。
* 信息的接收和发送均被禁止。
* 发送256+。

### CAN和CANFD的区别

* CAN = 8
* CANFD = 64
* CAN没有速率，CANFD有速率
* CAN FD: 向下兼容CAN，但传统CAN节点无法解码CAN FD帧。
* CAN: 数据阶段和仲裁阶段的速率相同（最高1 Mbps）。
* CAN FD: 数据阶段支持更高速率（最高可达8 Mbps），仲裁阶段保持1 Mbps。

### 当CAN总线没有报文时

#### 硬件检查

* 检查CAN收发器、电缆连接、终端电阻（120Ω是否正确安装）。确保电源供给正常。

#### 节点状态

* 确认所有节点上电并工作，检查是否有节点进入Bus Off状态。

#### 波特率匹配

* 确保所有节点的波特率一致。

#### 工具验证

* 使用CAN分析仪监听总线，确认是否有任意节点发送。

#### 软件配置

* 检查发送代码和滤波配置，确保报文未被屏蔽。

#### 测试发送

* 使用CAN分析仪或手动发送工具测试发送单帧，观察总线响应。

### BUSOFF

* 表示某个节点由于严重通信错误被系统隔离的一种状态。进入BUSOFF后会停止收发。(没有检测到显性位)

### 造成原因

* 错误计数器或接受错误计数器达到阈值
* ECU连续5次快恢复后，将执行慢恢复。
* 慢恢复在进入BUSOFF状态等待慢恢复时间后，再尝试向总线发送报文。

### **如何恢复**

* 等待自动恢复，有一个计时器等待一段时间，比如128次总线错误，则设备尝试重新连接到总线。
* 清除busoff错误
* 初始化CAN控制器
* 检查硬件是否错误

### 

## 信息安全

### AES

* CBC模式中，IV防止相同明文加密结果相同，使用IV可以确保即使相同的明文被加密多次，生成的密文也会不同

### 分类

* 对称加密：AES
* 非对称加密：RSA
* 哈希算法：SHA

## ISO15765

### 网络层协议

#### 协议功能

* 发送/接收最多4095个字节的数据信息。
* 报告发送/接收完成状态。

#### 单帧发送

* 最多发送6字节或（正常地址情况下）7字节数据，按照N\_PDU格式发送信息，称为单帧（SF）。

#### 多帧发送

* 长信息的发送通过拆分信息并通过多个N\_PDU发送的形式。
* 长信息的接收通过接收多个N\_PDUR并通过重组这些接受的数据。这多个N\_PDU包括首帧及连续帧。

##### BS

* 块大小：在授权继续发送其余的N\_PDU之前，接收方允许发送方最大的N\_PDU个数。

##### STmin

* 间隔最短时长：发送方在发送两个连续帧间隔等待的最短时间。

##### 流控帧（机制）

* 30 00 00:
  + 30：继续发送
  + 31：等待
  + 32：OverFlow
  + Byte2（BlockSize）：收到流控帧后,可发送的连续帧数量。设置为0,发送数量无限制
  + Byte3（Stmin/ms）：表示连续发送连续帧的间隔时间
* FC.CTS：继续发送
* FC.WAIT：请求继续等待
* FC.OVFLW：缓冲溢出

##### 

#### 网络层时间参数

##### N\_As

* 发送方任意帧类型从请求发送到发送完成 (接收到对应的ACK) 的时间间隔。一般Timeout时间设定在25ms。

##### N\_Ar

* 接收方任意帧类型从请求发送到发送完成的时间间隔。一般Timeout时间设定在25ms。

##### N\_Bs

* N\_Bs可理解为Sender从XXX到收到流控帧 的时间,XXX分为三种情况：
* Sender从发送完首帧（FF）到收到流控帧的时间；
* Sender从发送完连续帧（CF）到收到流控帧 的时间；
* Sender从收到流控帧（FC）（状态为Wait）到收到下一个流控帧的时间；

##### N\_Br

* N\_Br可理解为Receiver从XXX到请求发送流控帧的时间，XXX分为三种情况：
* Receiver从收到首帧（FF）到请求发送流控帧FC的时间；
* Receiver从收到连续帧（CF）到请求发送流控帧FC的时间；
* Receiver从发送完流控帧（FC）(状态为Wait) 到请求发送下一个流控帧的时间。

##### N\_Cs

* 发送方从XXX到请求发送连续帧（CF） 的时间,XXX分为两种情况
* Sender 从 收到流控帧（FC）到 请求发送连续帧（CF） 的时间
* Sender从 发送完连续帧（CF）到 请求发送下一包连续帧（CF） 的时间

##### N\_Cr

* 接收方从XXX到收到下一个连续帧（CF） 的时间,XXX分为两种情况
* Receiver 从发送完流控帧（FC）到收到连续帧（CF）的时间
* Receiver 从 收到连续帧（CF）到 收到下一包连续帧（CF） 的时间

### 应用层时间参数

#### P2 CAN\_Client

* 诊断成功发送诊断报文请求之后 到 收到ECU回复诊断响应的超时时间间隔

#### P2 CAN\_Server

* ECU接收到诊断请求之后到开始发送诊断报文的时间间隔

#### P2\* CAN\_Client

* 诊断接收到78后继续等待ECU响应的时间间隔

#### P2\* CAN\_Server

* ECU收到78后继续发送下一帧诊断响应报文的时间间隔，一般设置在5000ms

#### P3 CAN Client Phys

* 诊断成功发送物理寻址诊断请求之后，无需ECU回复继续发送下次物理寻址诊断请求的最小时间间隔

#### P3 CAN Client Func

* 诊断成功发送功能寻址之后再次发送功能寻址的最小时间间隔

#### 例子

* 发送诊断报文 -> P2 CAN\_Client定时器启动(收到ECU回复的响应时间)。
* 收到诊断报文后 -> 在P2 CAN\_Server时间内回复。
* 诊断收到78后（否定响应） -> 等待P2\* CAN\_Client时间（等待ECU响应）。
* P2\* CAN\_Server(ECU收到78后) 继续发送下一个响应消息的时间。

### 会话层时间参数

#### S3 Client

* 客户端定时参数。客户端保持非默认会话自动化连接；发送用于保持诊断会话的信息请求之间的时间。通常4000ms。

#### S3 Server

* 没有接收到任何请求信息时，服务器保持诊断会话的时间。通常5000ms。

## 

## 

# C语言

## 指针与其他

### 指针与常量

* const int \*p 指向常量的指针（不能修改 \*p，但 p 可以指向别的地址）
* int \*const p 指针本身是常量（p 不能变，但 \*p 可变）
* const int \*const p 指针和数据都是常量（p 和 \*p 都不能变）

**解释：**

const int a = 10;

const int \*p1 = &a;   // p1 可以改指向别的地址，但 \*p1 不能改

int b = 20;

int \*const p2 = &b;   // p2 不能改指向别的地址，但 \*p2 可以改

const int \*const p3 = &a;  // p3 既不能改指向，也不能修改 \*p3

**const int \*p1 = &a; p1的值永远是10，但是地址可以改变，改变之后也是10。**

**int b = 20; int \*const p2 = &b; 这个地方是p2不能指向别的地址，但是值可以改变。**

### 指针与数组

int arr[5] = {10, 20, 30, 40, 50};

int \*p = arr;  // p 指向 arr[0]

printf("%p %d\n", p, \*p);

*p是指针， \*p是数值*

### 什么是野指针？

* 指向未知地址或已释放地址的指针，应该NULL化或者free

### 

### 堆栈溢出

* 递归过深
* 局部变量过大
* 函数嵌套太多
* 栈空间分配

## 自增

#include <stdio.h>

int main()

{

int x = 5;

int a = x++;  // 先赋值 a = 5，然后 x 变为 6

int b = x++;  // 先赋值 b = 6，然后 x 变为 7

int c = x++;  // 先赋值 c = 7，然后 x 变为 8

printf("a = %d, b = %d, c = %d, x = %d\n", a, b, c, x);

return 0;

}

**函数的参数求值顺序是未定义的!这里 x++ 可能按不同顺序执行，例如：**

* **可能是 (5, 6, 7)**
* **可能是 (7, 6, 5)**
* **也可能是 (6, 5, 7)**

### 自增的副作用

int x = 5;

int a = x++;

等价于：

int a = x;  // 先把 x 的当前值（5）赋值给 a

x = x + 1;  // 然后 x 自增，变成 6

## 函数

### Malloc/Calloc

#### Malloc

##### 分配内存

* Malloc(size) 分配 size 字节，内容是随机值（未初始化）。

#### Calloc

##### 分配内存

* Calloc(n, size) 分配 n \* size 字节，并全部初始化为 0。

##### Free

* Free之后赋值NULL，防止野指针（Free后，指针仍然指向旧地址，可能会被误用）。

### Strlen

#### 作用

* 计算字符串的长度（直到\0终止符，不包括\0）。

## 大端/小端

### 大端字节序

* 高字节存储在低地址，低字节存储在高地址（高位在前，低位在后）。
* 0X12345678的存储方式如下（地址从0开始）：

地址 | 数据

--------------

0x00 | 0x12

0x01 | 0x34

0x02 | 0x56

0x03 | 0x78

### 小端字节序

* 低字节存储在低地址，高字节存储在高地址（低位在前，高位在后）。
* 0X12345678的存储方式如下（地址从0开始）：

地址 | 数据

--------------

0x00 | 0x78

0x01 | 0x56

0x02 | 0x34

0x03 | 0x12

### 

### 

## 堆栈

### 栈(Stack)

* 由系统自动分配，存储局部变量、函数参数、返回地址等。
* 后进先出的方式管理数据。
* 由编译器自动分配，存储局部变量、函数参数、返回地址等。
* 申请栈与释放：由编译器自动管理，当函数调用结束时，局部变量的栈空间自动释放。
* 作用范围：局部作用域（函数内），函数返回后变量自动释放。

### 堆(Heap)

* 手动分配与释放，适用动态存储分配的数据：数据、结构体、对象。
* 全局有效。

### 

## 

## 

## 

# Linux

## TCP/UDP

### TCP

#### 连接方式

* 面向连接，通信连接需要建立连接（三次握手），通信结束后需要断开连接（四次挥手）。

#### 可靠性

* 可靠传输，提供数据包确认、丢包重传、流量控制、拥塞控制，确保数据按序到达且不丢失。

#### 传输效率

* 由于要进行确认、重传和流控，传输速度较慢，但稳定。

#### 数据传输方式

* 面向字节流，数据是无结构的，发送端和接收端的数据可能被拆分或合并。

### UDP

#### 连接方式

* 无连接，直接发送数据，不需要建立和断开连接。

#### 可靠性

* 不可靠传输，不保证数据包是否到达、顺序是否正确，也不提供重传机制。

#### 传输效率

* 传输速度快。

#### 数据传输方式

* 面向数据包，每个数据包独立发送，应用层需要自己处理数据的顺序和完整性。

### UDP



## 

# 