

第七章. OBD 诊断协议在 Neulen TBOX C300 开发板中的程序实现

本章开始学习 OBD 诊断协议的解读以及在 C300 开发板中的代码实现。OBD 诊断协议相对来说是一类相对容易的 OSI 模型的协议。我个人按照他们的执行程度以及定义的完整性分成了四类：

第一类，执行程度最高，有完整的物理层到应用层定义，且乘用车不执行这类协议不允许销售，它们是排放协议（Emission-Related）。顾名思义就是检测尾气排放的协议，但是它的参数值跟我们理解的污染尾气二氧化碳，二氧化硫一点关系的都没有。它的英文缩写北美和中国叫 OBDII，咱们把它叫成 OBD2，在日本叫 JOBD，在欧洲叫 EOBD。所以看到如上缩写我们就知道它们执行的标准就是排放协议。这些协议名称是 ISO15765-4, ISO14230-4, ISO9141-2, SAEJ1850. 以上协议都完成了 OSI 模型上的总定义。怎么理解？这些协议都有自己的 OSI 模型分层定义，了解其中一个协议你会接触到很多个 ISO 或者 SAE 协议分别定义它们的物理层，数据链路层，会话层，应用层。但是之所以叫这类协议是排放协议它们在最上层的应用层是统一的，都采用了 ISO15031 协议。但是不要认为在 ISO15031 找到的定义数据就一定能从某辆车获得，车厂会选择其中一部分执行。举个最简单的例子，我们使用 OBD2 读车架号，发现一些车能读到，而另一些车不能读。不能读那是汽车厂偷懒了。

第二类，只在大部分商用车和农用机械上使用，执行程度不算得高，有完整的物理层到应用层定义，但是应用层并不强制执行，车厂可以根据自己实际情况定义自己的应用层。这就是 SAEJ1939，所以你会奇怪一些商用车明明是 SAEJ1939-21 协议数据格式，但是怎么也和 SAEJ1939-71 应用层对应不上。

第三类，不管乘用车还是商用车都有用这个协议，定义完整物理层到会话层，但是最上层的应用层只定义了一部分，并且定义的这部分还允许车厂看着办。这就是 UDS 协议，中文叫“统一诊断服务协议”。协议名称 ISO14229。这个协议我用一句俗语概括：“理想很美好，现实很骨干”。为什么？最后我会告诉你原因。

第四类，物理层到会话层，基本按照相关 ISO 或者 SAE 定义进行，但是应用层完全车厂自行定义。这类协议包括第三类和部分第二类协议应该叫车厂自定义诊断协议，网上有人起名专车协议或者叫私有协议。这一类我是觉得整车厂最喜欢的一类协议了。

最后，仅仅个人浅薄观点认为整车厂为什么更喜欢自定义协议。因为整车厂更希望在故障诊断上拥有最权威话语权以保障其授权经销商的售后利益，以及保证原厂配件的供应不受副厂抢单。所以做到统一诊断协议是几乎不可能的。这也为诊断开发的工程师提供了用武之地，仅个人浅薄之见，不接受反驳和赞同意见，不用找我聊这个事，我时间比较紧张。下面看具体协议解读和实现办法。

7.1 SAEJ1939 协议以及在开发板中的程序实现

7.1.1 SAEJ1939 协议解读

我们解读协议的办法和纯学习的看协议不一样，我们会带着工程项目的目地去解读协议，获取对我们项目有用的关键信息并加以实践。所以在解读之前我们必须提出我们要在协议中获取那些信息，也就是完成我们 SAEJ1939 诊断端设备的程序编写需要解决那些关键问题。下面我们就罗列出它们。

1. 协议通信如何唤醒？
2. 数据格式？
3. 信息通信管理？
4. 应用层如何定义？

1. 协议通信如何唤醒？

首先看第一点 协议通信如何唤醒，这个问题不需要找资料，因为物理层基于 CAN 通信的都不需要唤醒或者叫初始化。因为 CAN 是一种广播总线。

2. 数据格式？

一个 CAN 帧数据能装载应用的地方无非只有 CAN 标识符和数据域。关于这部分定义内容可从数据链路层协议 SAEJ1939-21 获得。以下讲解的内容和截图均出自这个协议。

看 SAEJ1939-21 协议的 5.2 Protocol Data Unit(PDU)，这里我们就记住一个名词缩写 PDU 的意思就是协议数据单元，也就是 SAEJ1939 最小数据结构。

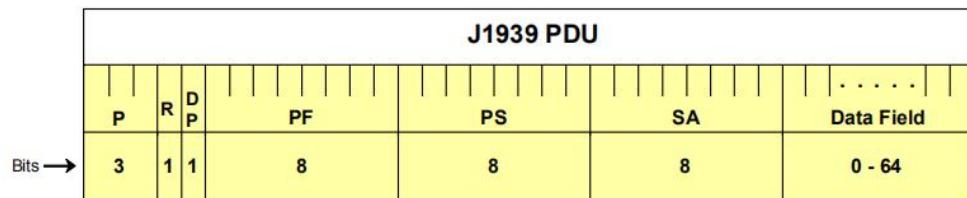


FIGURE 3 PROTOCOL DATA UNIT

这是从 SAEJ1939-21 FIGUER3 截图，它清晰的描画了一个 PDU 结构。首先看 CAN 标识符组成：

P：优先级，占用 CAN 标识符前 3 位，作用是决定该 PDU 进入总线的优先权限。值越小则优先级越高，所以 000 优先级是最高的 111 是最小的。这个其实我们可以不用理睬，因为在应用层里定义的数据自然会把它定义具体值。我们的 C300 开发板也无需处理这个值。

R: 预留的，占用 CAN 标识符第 4 位。如果您自己设计一套系统，假如您是航天部门的，设计一个用 CAN 通信的局域网用的就是 J1939 协议的话，PGN 已经超过了 131071。那么你就就可以考虑用这个预留位了。因为除了航天部门我想不出哪里可以用到 131071 个 PGN 了。这里有个新名词 PGN 全称 Parameter Group Number。中文 参数组号码。当前您要理解这个 PGN 是什么我只能类比，如果您熟悉 ISO 诊断协议的话 PGN 等同于 SID 的作用，如果您对 ISO 诊断协议也不了解，那么 PGN 就是一个功能索引，比如有车速 水温等数据，车速对应一个 PGN，水温对应一个 PGN。但是也不绝对，也许车速和水温对应一个 PGN，这在应用层协议解读上我们再探讨，您只需要知道 PGN 算是一个功能的索引吧。

DP: 数据页，占用 CAN 标识符第 5 位。这是 PGN 装载的一个位置，当然 PGN 超过 65536 后该位才会被置 1。

PF: PDU 格式 (PDU Format)，占用 CAN 标识符第 6 位到 13 位，这也是 PGN 装载的位置，但是装载的具体值是否大于等于 240 决定了 PDU 的格式，PDU 格式分为 PDU1 和 PDU2。PDU1 和 PDU2 区别就在于下一个标识 PS 装载的内容。所以 关于 PDU1 和 PDU2 格式我们稍后具体讲解。

PS: PDU 特征 (PDU Specific)，占用 CAN 标识符第 14 位到 21 位，当 PF 决定格式是 PDU1 时该域存放的是目标地址 (DA:Destination Address)，如果 PF 决定格式是 PDU2 时，该域存放 PGN 扩展 (GE:Group Extension)。

SA: 源地址 (Source Address)，占用 CAN 标识符第 22 位到第 29 位。源地址 (SA) 和目标地址 (DA) 是相对而言的。SAEJ1939 定义了完整的地址值，比如发动机控制单元的地址是 00，汽车诊断设备也就是我们编写诊断程序的设备比如我们的 Neulen TBOX C300 开发板地址就应该是 F9。在哪份协议中定义呢？SAEJ1939 协议 这是一份总览协议. 从目录中找到 Address and Identity Assignments (地址和标识分配的表格，这里有具体定义)。如下图所示。

Table B2
J1939 Preferred Addresses
Industry Group #0 – Global

Note: Preferred Addresses 128 thru 247 are Industry Group specific. See Tables B3 thru B9.

Rev	SA	Controller Application	Comments	Associated NAME Function
	0	Engine #1	The #1 on the Engine CA is to identify that this is the first PA being used for the particular function, Engine. It may only be used for the NAME Function of 0, Function Instance 0, and an ecu instance of 0, which is commonly known as the "first engine".	0
	1	Engine #2	The #2 on the Engine CA is to identify that this is the second PA available for use for the function, Engine. It may be used by the "second" engine (Function 0, Function Instance 1, ECU Instance 0), but it may also be used by the second ecu on the first engine (Function 0, Function Instance 0, ECU Instance 1), if there is no second engine.	0
	3	Transmission #1	The first transmission - may only be used for the NAME Function of 3, Function Instance 0, and an ecu instance of 0.	3
	4	Transmission #2	The second PA available for use for the function, Transmission. It may be used by the "second" transmission (Function 3, Function Instance 1, ECU Instance 0), but it may also be used by the second ecu on the first transmission (Function 3, Function Instance 0, ECU Instance 1), if there is no second transmission.	3
	5	Shift Console - Primary	The shift console mounted in the normal drivers position	5
	6	Shift Console - Secondary	A shift console mounted remotely from the normal drivers position (May not be used for any ecu instances of the primary shift console)	5

从 SA 这一列标注的就是相关应用的地址。当然 SA 对应的不是物理上的硬件，对应的是 CA(控制应用程序)。每一个 CA 对应一个 SA。这就是为什么有时候一个控制单元硬件可以有两个或者多个 SA，因为在这个控制单元里面编写了多个控制应用程序 (CA:Controller Application)。下面是 C300 开发板等诊断设备的地址截图也是在同一个协议一个表中，因为太长，大家可以打开配套资料进行查询。C300 开发板地址 F9 转换为十进制是 249. 定义为板外诊断服务工具。

249	Off Board Diagnostic-Service Tool #1	The address for the first off board diagnostic service tool - may only be used for the NAME Function of 129, Function Instance 0, and an ecu instance of 0.	129
-----	--------------------------------------	---	-----

以上就是 SAEJ1939 一个 PDU 的 CAN 标识符的基本结构，注意，PDU 构成还有数据域 (Data Field)，那么数据域是搭载具体信息的，比如故障码，数据流，车架号等信息，这些我们要到应用层协议再具体解读，但是数据域有一个很关键的信息我们一定要获取到，是什么信息呢？假设我们需要 SAEJ1939 的 PDU 搭载一个两个字节的数据在第一和第二字节。值是 65123，把这个值转换成十六进制是 0xFE63。好的 问题来了？ 数据域第一个字节放 FE 还是放 63 呢？下面是 SAEJ1939-21 的要求。

FIGURE 7-1 PDU FORMATS

5.4 Message Types

There are five message types currently supported. These types are the following: Commands, Requests, Broadcasts/Responses, Acknowledgment, and Group Functions. The specific message type is recognized by its assigned Parameter Group Number. Refer to Appendix A of SAE J1939 for examples of PGN assignments. The RTR bit (defined in the CAN protocol for remote frames) is not to be used in the recessive state (logical 1). Therefore, Remote Transmission Requests (RTR = 1) are not available for use in SAE J1939.

For multibyte parameters that appear in the data field of a CAN Data Frame they shall be placed least significant byte first. Exceptions are noted where applicable (i.e. ASCII data). So if a 2-byte parameter were to be placed in bytes 7 and 8 of the CAN DATA Frame, the LSB would be placed in byte 7 and the most significant byte in byte 8.

在 5.4 Message Types(信息类型)中描述，对于 CAN 数据域中出现的多参数，它们应该让低有效的字节优先，除了标志为 ASCII 编码的数据外。所以有两个字节的参数被放置于 CAN 数据域的第 7 和第 8 字节，那么低字节被放在第 7 字节而代表高字节就被放在第 8 字节。

所以这点很重要，我们刚才的假设 0xFE63 被搭载在 CAN 数据帧的数据域的第一和第二字节的话，那么排列方式应该是这样的 0x63 0xFE 低字节在前，高字节在后。这与 ISO 协议是不一样的。同时，对于 ASCII 编码的数据，举个例子比如 VIN 码，那么数据域就不会这种排列方式了，这时候高字节应该在第一个字节，低字节应该在最后一个字节。

另外从 FIGER3 图中可以看到，数据域是 0 到 64 位，也就是有 8 个字节一个 PDU 数据域，对于刚才我们举例子的 VIN 码是 17 个字节的，这时候，显然一个 PDU 无法装载。就需要多帧处理。我们将会在 “信息通信管理？” 具体解读。

这里还有一个问题就是 PDU1 和 PDU2 的定义？

TABLE 3 PDU SPECIFIC

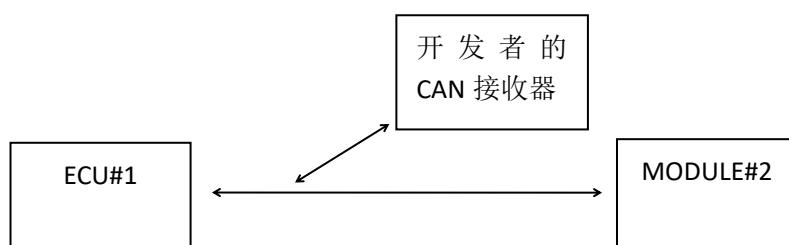
PDU Format Field	PDU Specific Field
PDU1 Format	0-239
PDU2 Format	240-255

从表格中很清晰的知道，PDU1 格式取决于 PF 值在 0 到 239 的话，PS 应该放目标地址（DA）。PDU2 格式取决于 PF 值在 240 到 255 的话，PS 应该放 PGN 扩展信息，其实这就是 PGN 低字节的值。所以站在应用层角度看，如果当前您要发送一个 PGN 大于或者等于 61440（F000）的值您就必须采用 PDU2 格式进行发送，小于 61440 的话就采用 PDU1 发送。

3. 信息通信管理

首先要明确的是通信速率，关于通信速率的定义可在 SAEJ1939-11 协议中找到，SAEJ1939 物理通信速率是 250K。

其次在进行具体信息通信管理解读之前还要明确下，对于诊断协议是请求形式还是广播形式。当然是请求形式。那么为什么侦听总线可以获得标准的 SAEJ1939 协议呢？这里我们要知道这个协议它并不仅仅用于诊断，同时它会存在车内各个电控系统间通信的情况，如下图所示：



对于大多数的车辆来说，无论是发动机 ECU，车身系统，ABS 系统等等这些电控系统和诊断 OBD 接口总线是挂在同一根 CAN 总线上的。也就出现了一种现象，当您在 CAN 接口接入 CAN 接收器的时候会发现 CAN 数据汹涌而至。尤其是对于采用 SAEJ1939 协议的商用车 OBD 接口，从中可以发现一些数据是 SAEJ1939-71 所定义的。所以对于初学者认为，OBD 诊断是不需要请求可以直接获得的，这是一种错误的认识，这种认识可能导致研究方向相偏差迟迟做不出产品，或者功能无法满足而项目停止。

我这里告诉一个原因，为什么汽车诊断 OBD 都是请求形式？这是 CAN 总线物理结构先天性缺陷造成的。CAN 总线虽然号称是广场总线，但是这种广场效应是通过提高通信速率的一种假象。CAN 总线依然是一种通过高低电压变化实现数字信号的串行通信线，既然是串行通信线那么在一个时间单位内只能发送一个 CAN 数据帧。而对于汽车 OBD 诊断设备，尤其是专业诊断设备，一个车型仅仅数据流的通信数据量就很有可能有几千条，如果把这些诊断数据全部做成广播形式，那就直接加重总线负荷，造成一些汽车安全数据的延时，这是致命性的，比如主动安全信号。

那么汽车在设计诊断系统的时候,是不是把所有数据都做成请求的形式呢?乘用车的答案是肯定的,但是基于 SAEJ1939 不一样,有部分数据做成广播的形式,但是其设计的目的并不是给诊断设备直接从 OBD 获取的,它的目的就像上图所示, MODULE#2 需要 ECU#1 的一个数据,比如车速。打个更形象的例子, ECU#1 是发动机控制单元,MODULE#2 是仪表台对应的某个电控单元。好了, MODULE#2 需要获得发动机控制单元的发动机转速数据,怎么办?反正诊断系统有这个数据,我就把它开放成广播的形式,给 MODULE#2 获取,或者 MODULE#2 请求,再或者不止 MODULE#2 一个电控单元要这个数据,太多电控单元要用,我就把它的目标地址做成全局方式在总线上广播。所以,如果您设计 OBD 产品时,没有通过请求就能获得了相关数据那么说明您很幸运。如果找不到相关数据,建议您老实通过请求命令请求并等待汽车内的诊断系统响应。

如何创建一个 SAEJ1939 协议的请求命令呢?如图所示,这是在 SAEJ1939-21 协议的截图。

Parameter Group Name:	Request
Definition:	Used to request a Parameter Group from a network device or devices.
Transmission repetition rate:	Per user requirements, generally recommended that requests occur no more than 2 or 3 times per second.
Data length:	3 bytes (The CAN frame for this PG shall set the DLC to 3.)
Data page:	0
PDU Format:	234
PDU specific field:	Destination Address (global or specific)
Default priority:	6
Parameter Group Number:	59904 (00EA00 ₁₆)
Byte: 1,2,3	Parameter Group Number being requested (see Section 5.1.2 for field definition and byte order)

严格按照上面截图所要求内容,我们尝试创建一个 C300 开发板读取故障码的请求命令。第一步明确源地址和目标地址,对于 C300 开发板可以以诊断工具的身份接入 SAEJ1939 的 CAN 总线。前面关于帧格式的相关描述我们知道,诊断工具可以以 249 源地址接入总线。C300 开发板是以 249 源地址的身份接入的,源地址的十六进制就是 F9。接下来要明确目标地址,目标地址可以是全局目标地址 (DA Global) 或者特定目标地址(DA Specific),比如发动机 CA (控制应用) 地址 0。选全局目标地址还是特定目标地址主要是您希望响应的数据在遇到多帧的时候是以 BAM 模式还是以 RTS/CTS 模式通信。这个有点复杂,只要看懂下面的 SAEJ1939-21 截图。

TABLE 4 PDU1 AND PDU2 TRANSMIT, REQUEST AND RESPONSE REQUIREMENTS

PDU Format of Requested PGN	Data Length of Requested PGN	Request PGN 59904	Response	TP Used
1	≤ 8 bytes	DA Specific	DA Specific	NA
1	≤ 8 bytes	DA Global	DA Global	NA
1	≤ 8 bytes	none	DA Global	NA
			DA Specific	NA
1	> 8 bytes	DA Specific	DA Specific	RTS/CTS
1	> 8 bytes	DA Global	DA Global	BAM
1	> 8 bytes	none	DA Global	BAM
			DA Specific	RTS/CTS
2	≤ 8 bytes	DA Specific	DA Global	NA
2	≤ 8 bytes	DA Global	DA Global	NA
2	≤ 8 bytes	none	DA Global	NA
2	> 8 bytes	DA Specific	DA Specific	RTS/CTS
2	> 8 bytes	DA Global	DA Global	BAM
2	> 8 bytes	none	DA Global	BAM
			DA Specific	RTS/CTS

这里知道看协议要看英文版的重要性，在网上有很多翻译成中文版的 SAEJ1939 协议，在我做 SAEJ1939 前就有朋友发了一份 SAEJ1939-71 中文协议给我，问我能不能做，但是我看了之后发现有很多地方不知道具体描述什么？对照原版后才知道其具体意义。同时也发现了不少标错的地方。所以尽管看原版协议很困难，但是克服这个语言困难获得的信息是成逻辑和便于理解的。上图中的第一列和第二列我们可以截图网上翻译的中文版本协议进行对照如下图所示。

表 4 — PDU1 和 PDU2 传输, 请求和响应要求

PDU 长度	数据 长度	请求 PGN 59904	响应	使用传输协 议
1	≤8 字节	特定 DA	特定 DA	NA
1	≤8 字节	全局 DA	全局 DA	NA
1	≤8 字节	无	全局 DA	NA
			特定 DA	NA
1	> 8 字节	特定 DA	特定 DA	RTS/CTS
1	> 8 字节	全局 DA	全局 DA	BAM
1	> 8 字节	无	全局 DA	BAM
			特定 DA	RTS/CTS
2	≤8 字节	特定 DA	全局 DA	NA
2	≤8 字节	全局 DA	全局 DA	NA
2	≤8 字节	无	全局 DA	NA
2	> 8 字节	特定 DA	特定 DA	RTS/CTS
2	> 8 字节	全局 DA	全局 DA	BAM
2	> 8 字节	无	全局 DA	BAM
			特定 DA	RTS/CTS

PDU Fomat of Requested PGN 被错误翻译成 PDU 长度。

Data Length of Requested PGN 被错误翻译成 数据长度。

所以看着中文翻译协议您总会一头雾水，PDU Fomat of Requested PGN 正确意思就是 被请求的 PGN 的 PDU 格式。Data Length of Requested PGN 正确意思是 被请求的 PGN 的数据长度。这里注意 Requested 和 Request 的区别，前者是被请求后者是请求。所以再透过第三列 Request PGN 59904（请求 PGN 0xEA00）就知道正确的理解这个表格意思。PDU Fomat of Requested PGN 理解为“响应数据的 PGN 的 PDU 格式”，而 Data Length of Requested PGN 理解为响应数据的数据长度。如下图所示，在 PGN 定义中，PGN 的 BYTE2 字节代表的就是 PF，所以 PDU Fomat of Requested PGN 真正意义就是被请求数据的 PGN 第二字节的判断，大于等于 240 就是 2，否则就是 1。

TABLE 2 PARAMETER GROUP NUMBER EXAMPLES

PGN Constituent Components PGN (MSB) Byte 1 Sent third in CAN Data Frame	PGN Constituent Components PGN (MSB) Byte 1	PGN Constituent Components PGN (MSB) Byte 1	PGN Constituent Components PGN Byte 2 Sent second in CAN Data Frame	PGN Constituent Components PGN (LSB) Byte 3 Sent first in CAN Data Frame	Number of Assignable PGs	Cumulative Number of PGs	SAE or Manufacturer Assigned
Bits 8-3	R Bit 2	DP Bit 1	PF Bits 8-1	PS Bits 8-1	PGN Dec ₁₆	PGN Hex ₁₆	
0	0	0	0	0	0	000000 ₁₆	239
0	0	0	238	0	60928	00EE00 ₁₆	
0	0	0	239	0	61184	00EF00 ₁₆	1
0	0	0	240	0	61440	00F000 ₁₆	240
							SAE
0	0	0	240				
0	0	0	254	255	65279	00EFFF ₁₆	3840
0	0	0	255	0	65280	00FF00 ₁₆	4080
0	0	0	255				MF
0	0	1	0	255	65535	00FFFF ₁₆	4336
0	0	1	0	0	65536	010000 ₁₆	
0	0	1	239	0	126720	01EF00 ₁₆	240
0	0	1	240	0	126976	01F000 ₁₆	4576
0	0	1	255	255	131071	01FFFFFF ₁₆	4096
TOTALS							SAE
							8672
							8672

所以回到正题，我们想通过 C300 开发板创建一个故障码请求命令，请求的目标地址是全局目标地址还是特定目标地址，取决于我们希望响应的数据在遇到多帧的时候是以 BAM 模式还是以 RTS/CTS 模式通信。这里对照上表给出三种情况。但是在此之前先提前告诉大家一个知识点，故障码的 PGN 是 0xFECA。按照 PGN 定义是 24 位的，那么它的 BYTE2 就是 FE（十进制是 254），也就是大于 240，所以无论如何，PDU Format of Requested PGN 是 2。

情况 1：汽车无故障码或者故障码通过一帧数据就能完成响应。Data Length of Requested PGN $\leq 8\text{byte}$ 。所以无论请求的目标地址是全局目标地址（DA Global）还是特定目标地址（DA Specific）被请求 PGN 的响应数据都会发送到全局目标地址（DA Global）。并且不使用传输协议（TP:Transport Protocol）如下截图。

TABLE 4 PDU1 AND PDU2 TRANSMIT, REQUEST AND RESPONSE REQUIREMENTS

PDU Format of Requested PGN	Data Length of Requested PGN	Request PGN 59904	Response	TP Used
1	$\leq 8\text{ bytes}$	DA Specific	DA Specific	NA
1	$\leq 8\text{ bytes}$	DA Global	DA Global	NA
1	$\leq 8\text{ bytes}$	none	DA Global	NA
			DA Specific	NA
1	$> 8\text{ bytes}$	DA Specific	DA Specific	RTS/CTS
1	$> 8\text{ bytes}$	DA Global	DA Global	BAM
1	$> 8\text{ bytes}$	none	DA Global	BAM
			DA Specific	RTS/CTS
2	$\leq 8\text{ bytes}$	DA Specific	DA Global	NA
			DA Global	NA
2	$\leq 8\text{ bytes}$	none	DA Global	NA
			DA Specific	RTS/CTS
2	$> 8\text{ bytes}$	DA Global	DA Global	BAM
2	$> 8\text{ bytes}$	none	DA Global	BAM
			DA Specific	RTS/CTS

情况 2：汽车故障码响应数据超过 8 个字节，需要多帧数据响应，且请求命令通过特定目标地址（DA Specific）请求。这时候响应数据将发送到特定目标地址，也就是诊断设备源地址 F9。并且使用传输协议 RTS/CTS 模式，如下截图所示。

TABLE 4 PDU1 AND PDU2 TRANSMIT, REQUEST AND RESPONSE REQUIREMENTS

PDU Format of Requested PGN	Data Length of Requested PGN	Request PGN 59904	Response	TP Used
1	$\leq 8\text{ bytes}$	DA Specific	DA Specific	NA
1	$\leq 8\text{ bytes}$	DA Global	DA Global	NA
1	$\leq 8\text{ bytes}$	none	DA Global	NA
			DA Specific	NA
1	$> 8\text{ bytes}$	DA Specific	DA Specific	RTS/CTS
1	$> 8\text{ bytes}$	DA Global	DA Global	BAM
1	$> 8\text{ bytes}$	none	DA Global	BAM
			DA Specific	RTS/CTS
2	$\leq 8\text{ bytes}$	DA Specific	DA Global	NA
2	$\leq 8\text{ bytes}$	DA Global	DA Global	NA
2	$< 8\text{ bytes}$	none	DA Global	NA
			DA Specific	RTS/CTS
2	$> 8\text{ bytes}$	DA Specific	DA Specific	RTS/CTS
			DA Global	BAM
2	$> 8\text{ bytes}$	none	DA Global	BAM
			DA Specific	RTS/CTS

情况 3：汽车故障码响应数据超过 8 个字节，需要多帧数据响应，且请求命令通过全局目标地址（DA Global）请求。这时候响应数据将发送到全局目标地址(0xFF)，并且使用传输协议 BAM。如下截图所示。

TABLE 4 PDU1 AND PDU2 TRANSMIT, REQUEST AND RESPONSE REQUIREMENTS

PDU Format of Requested PGN	Data Length of Requested PGN	Request PGN 59904	Response	TP Used
1	≤ 8 bytes	DA Specific	DA Specific	NA
1	≤ 8 bytes	DA Global	DA Global	NA
1	≤ 8 bytes	none	DA Global	NA
			DA Specific	NA
1	> 8 bytes	DA Specific	DA Specific	RTS/CTS
1	> 8 bytes	DA Global	DA Global	BAM
1	> 8 bytes	none	DA Global	BAM
			DA Specific	RTS/CTS
2	≤ 8 bytes	DA Specific	DA Global	NA
2	≤ 8 bytes	DA Global	DA Global	NA
2	≤ 8 bytes	none	DA Global	NA
2	> 8 bytes	DA Specific	DA Specific	RTS/CTS
2	> 8 bytes	DA Global	DA Global	BAM
2	> 8 bytes	none	DA Global	BAM
			DA Specific	RTS/CTS

所以我们可以读取故障码，可以很好举例 SAEJ1939 数据链路层对于数据传输的三种形式。单帧传输，多帧传输 BAM 模式和 RTS/CTS 模式。

图表我们看懂了，就解决了请求地址和响应地址怎么确定的问题和数据传输的形式问题。

Parameter Group Name:	Request
Definition:	Used to request a Parameter Group from a network device or devices.
Transmission repetition rate:	Per user requirements, generally recommended that requests occur no more than 2 or 3 times per second.
Data length:	3 bytes (The CAN frame for this PG shall set the DLC to 3.)
Data page:	0
PDU Format:	234
PDU specific field:	Destination Address (global or specific)
Default priority:	6
Parameter Group Number:	59904 (00EA00 ₁₆)
Byte: 1,2,3	Parameter Group Number being requested (see Section 5.1.2 for field definition and byte order)

创建 C300 开发板读故障码请求命令，组成元素如下：

1. 优先级(Default priority):6;
2. 数据页(Data Page): 0;
3. PDU 格式(PDU Fomat): 234(0xEA)
4. 全局目标地址(DA Global):255(0xFF)
5. 特定目标地址(DA Specific):0(发动机控制单元的源地址为例)
6. 源地址(Source Address): 0xF9
7. PGN: 0xFECA

同时，我们看到请求命令的 PF =234，是小于 240 的，所以采用 PDU1 格式进行构建命令。根据 SAEJ1939-21 协议的格式结构，如下图所示。

TABLE 5 USE OF THE SPECIFIED FIELDS IN SAE J1939S PDU1 FORMAT

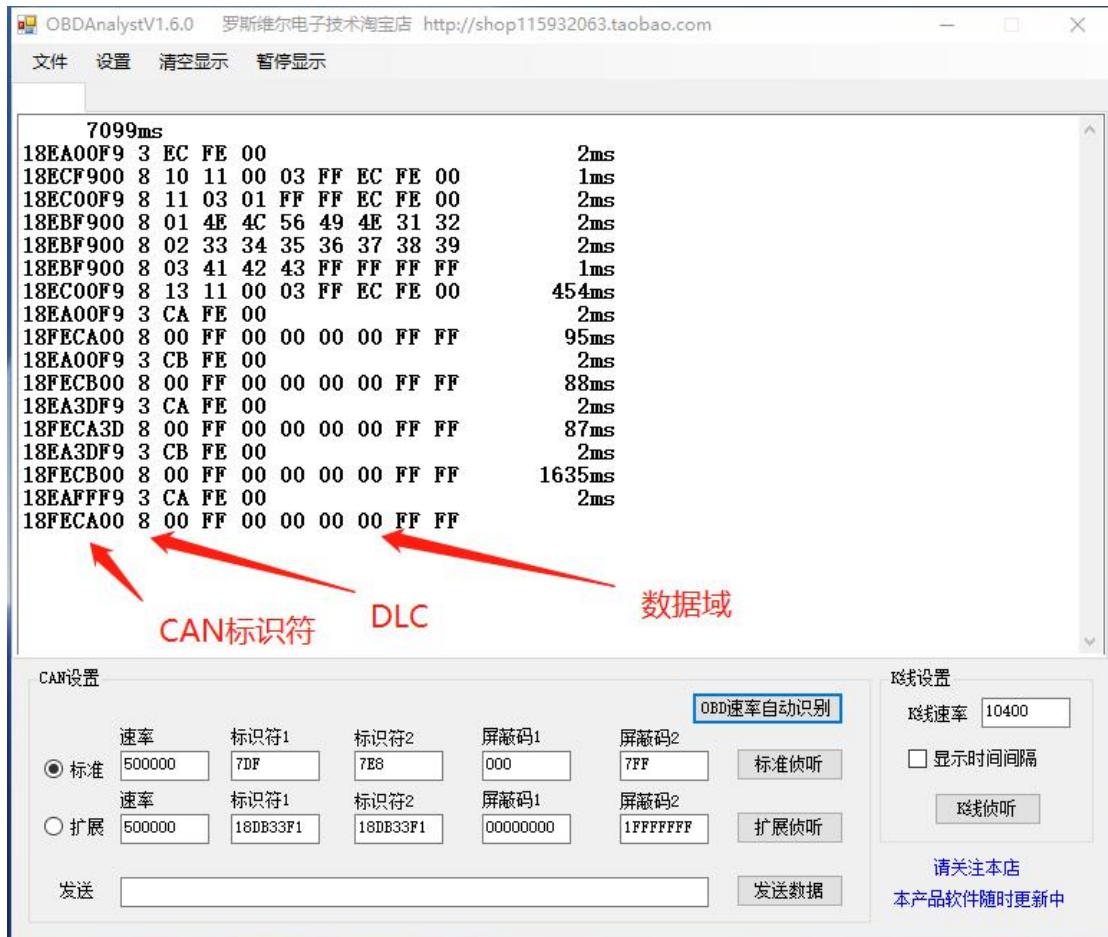
Message Type	PF	PS (DA)	SA	Data 1	Data 2	Data 3
Global Request	234	255 Responders	SA1 Requester	PGN LSB ¹	PGN	PGN MSB ¹
Specific Request	234	SA2 Responder	SA1 Requester	PGN LSB ¹	PGN	PGN MSB ¹

¹ The Parameter Group Number in the data field is used to identify the information being requested.

最终构建命令如下：

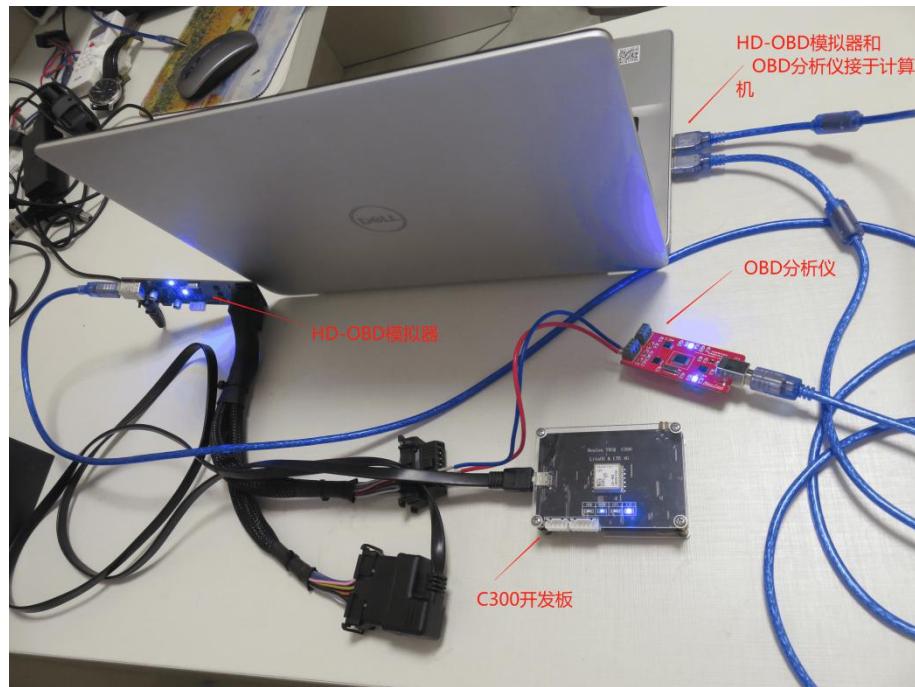
1. 以全局目标地址为目标地址的故障码请求命令: 18EAFFF9 3 CA FE 00
2. 以特定目标地址为目标地址的故障码请求命令: 18EA00F9 3 CA FE 00

这里的命令结构我是以我们 Neulen 的产品通用的一种结构来构建的，也就是我们 OBD 分析仪采集数据的一种结构（CAN 标识符+DLC+数据域）如下图所示。



前面我们已经提到了，故障码读取，看故障码响应数据可以很好举例 SAEJ1939 数据链路层对于数据传输的三种形式。单帧传输，多帧传输 BAM 模式和 RTS/CTS 模式。

这时候我们可以接上 C300 开发板连接于模拟器，同时并联上我们的 OBD 分析仪，如图所示。

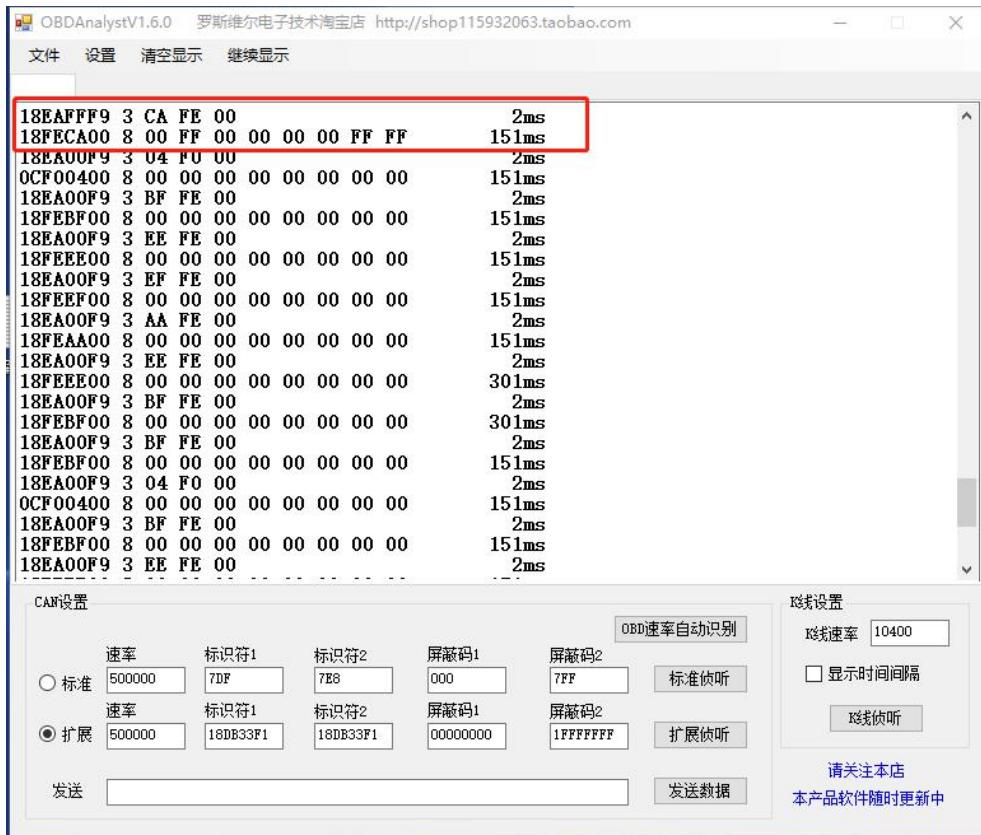


下面通过分别设置模拟器故障码，来分析单帧传输，多帧传输 BAM 模式和 RTS/CTS 模式。

单帧传输



模拟器设置无故障码，这时候 C300 开发板对模拟器发送故障码请求命令，如下图所示。



C300 开发板发送 以全局目标地址为目标地址的故障码请求命令: 18EAFFFF9 3 CA FE 00, 响应数据是 PDU2 格式的数据, PGN 存放于 PF 和 PS 的区域, 所以不存在目标地址, 这种情况默认是响应给全局目标的。

下面再看以特定目标地址为目标地址的故障码请求命令: 18EA00F9 3 CA FE 00, 如下图所示。



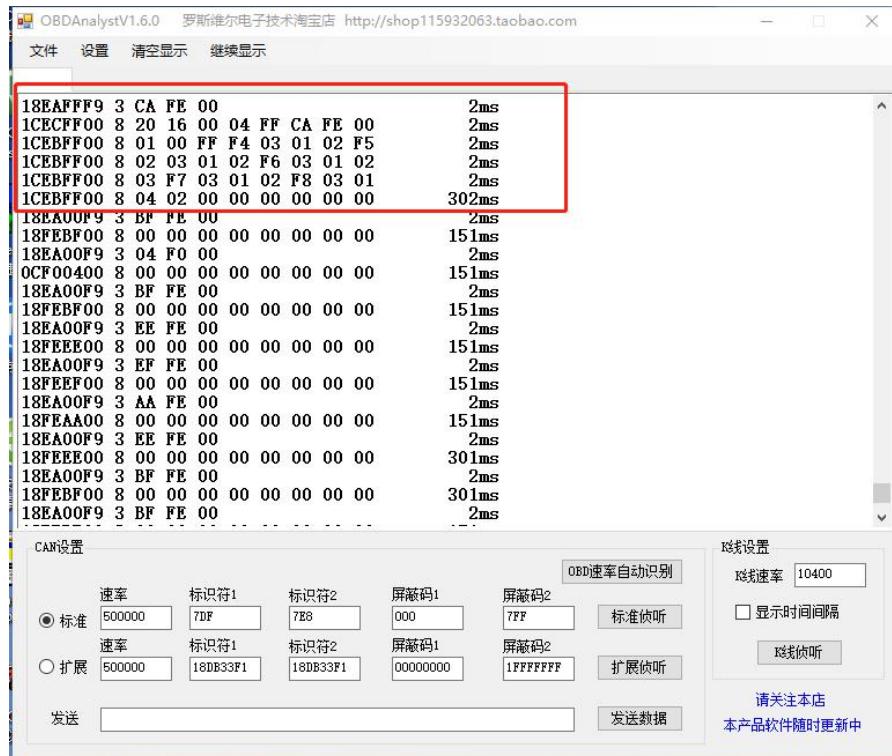
其响应数据可以看出，不管是使用以全局目标地址为目标地址的故障码请求命令还是以特定目标地址为目标地址的故障码请求命令，它们的响应数据都是一样的，因为响应数据的 PGN 大于 0xf000, 所以必须采用 PDU2 格式，PDU2 格式的数据没有目标地址可言，所以它们默认发送给全局目标地址。

多帧传输 BAM 模式

在分析数据前，先设置模拟器故障码，如下图所示，任意设置 5 个故障码。



这时候，C300 开发板发送 以全局目标地址为目标地址的故障码请求命令: 18EAFFFF9 3 CA FE 00。前面我们已经详细解读过协议，因为 PDU Format of Requested PGN 是 2，同时响应的数据大于 8 个字节，请求命令是全局目标地址，这三个条件决定了，响应数据必须以 BAM 模式进行响应，响应数据如下截图所示。



那么这里着重解读 BAM 模式的数据结构，请看下图 SAEJ1939-21 协议截图。

Parameter Group Name:	Transport Protocol—Connection Management (TP.CM)
Definition:	Used for the transfer of Parameter Groups that have 9 bytes or more of data. A definition of each specific message defined as part of the transport protocol is contained in the Sections 5.10.3.1 through 5.10.3.5.
Transmission repetition rate:	Per the Parameter Group Number to be transferred
Data length:	8 bytes
Data Page:	0
PDU Format:	236
PDU Specific:	Destination Address
Default priority:	7
Parameter Group Number:	60416 (00EC00 ₁₆)
Data ranges for parameters used by this Group Function:	
Control byte:	0-15, 18, 20-31, 33-254 are Reserved for SAE Assignment
Total Message Size, number of bytes:	9 to 1785 (2 bytes), zero to 8 and 1786 to 65535 not allowed
Total Number of Packets:	2 to 255 (1 byte), zero not allowed
Maximum Number of Packets:	2 to 255 (1byte), zero through 1 are not allowed
Number of Packets that can be sent:	0 to 255 (1 byte)
Next Packet Number to be Sent:	1 to 255 (1 byte), zero not allowed
Sequence Number:	1 to 255 (1 byte), zero not allowed

(截图 1)

Broadcast Announce Message (TP.CM_BAM): Global Destination		
Byte:	1	Control byte = 32, Broadcast Announce Message
	2,3	Total message size, number of bytes
	4	Total number of packets
	5	Reserved for assignment by SAE, this byte should be filled with FF ₁₆
	6-8	Parameter Group Number of the packetized message

(截图 2)

截图 1 和截图 2 共同描述了 TP.CM_BAM 数据的每一位和每一个字节具体定义。这里借助分析仪采集到的 C300 开发板通过故障码请求命令（18EAFF9 3 CA FE 00），请求 HD-OBD 模拟器响应 5 个故障码的 BAM 数据进行一个举例对比，分别把上面截图的定义对号入座。看采集到的第一个响应数据：

1CECFF00 8 20 16 00 04 FF CA FE 00

这个 TP.CM_BAM 数据分为三部分，首先 1CECFF00 是 29bit CAN 标识符，其次 8 是 DLC，DLC 并不重要，只是说明数据域由 8 个字节组成，最后是数据域 20 16 00 04 FF CA FE 00。

1CECFF00 CAN 标识符在截图 1 对应关系：

1. 1C 对应 Default priority:7。
2. EC 对应 PDU Fomat 236 (0xEC)，同时 PGN 等于 60416 是一个意思。
3. FF 对应 PDU Specific: Destination Address。前面的协议解读我们已经明确，这时候目标地址为全局目标(Global Destination)，所以必须是 0xff。
4. 00 对应 PDU 格式协议中的源地址，00 代表发动机控制单元的源地址，具体定义前面已经解读了，在 SAEJ1939 协议中有对应的地址表格。

20 16 00 04 FF CA FE 00 数据域在截图 1 和截图 2 的对应关系：

1. 20 第一个字节，表示控制字节（Control byte）。截图 2 明确了 在 TP.CM_BAM 数据中，Control byte=32 其十六进制为 20。
2. 16 00 第二字节和第三字节，表示信息长度，字节个数（Total message size,number of bytes），也就是说，信息长度指的就是下面 TP.DT 数据所承载的信息长度，当前的数据是 HDOBD 模拟器模拟的故障码信息，一个故障码需要占用 4 个字节，5 个故障码就是 20 个字节，所以转换成十六进制就是 16，因为是两个字节表示，前面协议格式中我们已经解读了，在 SAEJ1939 中存在两个以上字节的数据，在 SAEJ1939 数据结构中，理应把低字节放在 CAN 数据的高字节，高字节放在 CAN 数据的低字节，所以 16 00 表示 20 个字节的信息长度。
3. 04 第四字节，表示数据包的个数（Total Number of packets），这里的数据包指的就是帧数，哪些帧的数量呢？就是接下来模拟器要发送出去的 TP.DT 数据的帧数，上面我已经提到了 TP.DT 就是承载信息的 CAN 帧，DT 是 Data Transfer 的缩写。这里需要 4 帧数据承载模拟器响应给 C300 开发板的 20 个字节故障码信息。为什么是 4 帧，接下来会具体讲 TP.DT 数据结构。
4. FF 第五字节，这个字节是保留给 SAE 定义的，也就是未定义，需要把它设置为 FF。这是截图 2 的意思。
5. CA FE 00 第六字节，第七字节和第八字节，表示 PGN。故障码 PGN 是 0x00FECA 所以填充 CA FE 00

然后再看下面的截图 3, 如果把 Neulen C300 开发板和 HD-OBD 模拟器对号入座的话, 截图 3 的 Originator Node 对应 HD-OBD 模拟器, 而 Responder Node 对应 Neulen C300 开发板。这里如果您咬文嚼字的话认为这个对应关系是错的, 不过我们搞的是工程, 变通和逻辑能力很重要, 发起和响应是相对而言的, 并且是有前提条件的。去除故障码请求命令, 而把时间点集中在 HD-OBD 模拟器响应数据这个过程, 可以把 HD-OBD 模拟器认为是发起点, 而 Neulen C300 开发板是响应点。

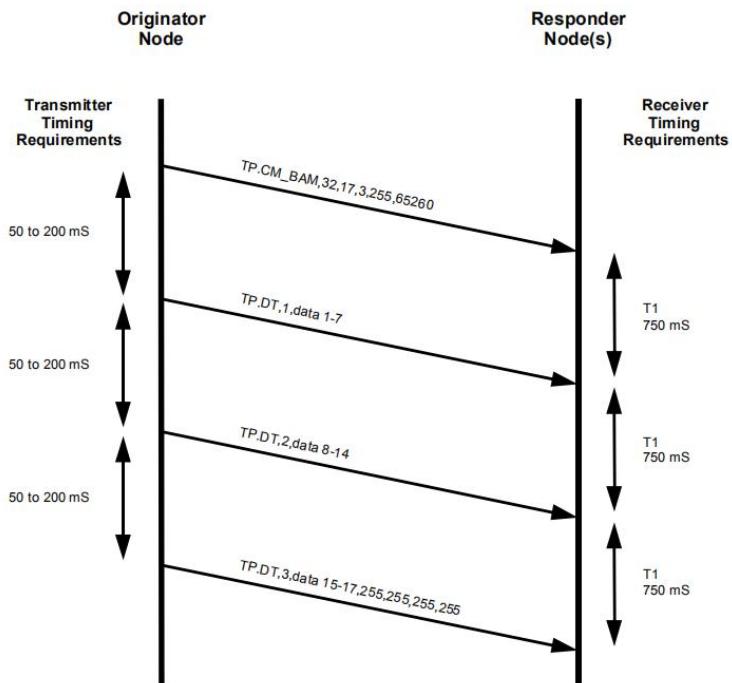


FIGURE C3 BROADCAST DATA TRANSFER

(截图 3)

我们看到 HD-OBD 模拟器响应 TP.CM_BAM 后, 会继续响应数据传输信息 (TP.DT), 关于 TP.DT 数据结构可以看下面截图 4 所示

Parameter Group Name:	Transport Protocol—Data Transfer (TP.DT)
Definition:	Used for the transfer of data associated with Parameter Groups that have more than 8 bytes of data
Transmission repetition rate:	Per the Parameter Group to be transferred
Data length:	8 bytes
Data Page:	0
PDU Format:	235
PDU specified field:	Destination address (Global (DA = 255) for TP.CM.BAM data transfers) (Global not allowed for RTS/CTS data transfers)
Default priority:	7
Parameter Group Number:	60160 (00EB00 ₁₆)
Data ranges for parameters used by this Group Function:	
Sequence Number:	1 to 255 (1 byte)
Byte:	1 2-8 Sequence Number Packetized Data (7 bytes). Note the last packet of a multipacket Parameter Group may require less than 8 data bytes. The extra bytes should be filled with FF ₁₆ .

FIGURE 15 TRANSPORT PROTOCOL—DATA TRANSFER MESSAGE (TP.DT)

(截图 4)

我们依然是看 OBD 分析仪采集的 HD-OBD 模拟器响应 Neulen C300 开发板的 TP.DT 数据，下图蓝色方框的内容就是 TP.DT 数据。



我们继续把数据对应协议截图 4 的定义对号入座，首先看 CAN 标识符 1CEBFF00

1. 1C 对应 Default priority: 7。
2. EB 对应 PDU Format 235 (0xEB), 同时对应 PGN :60160(00EB00)。
3. FF 对应 PDU Specific 此处一样是全局目标地址 所以填充 0xFF。
4. 00 对应源地址，发动机控制单元 SA 等于 0。

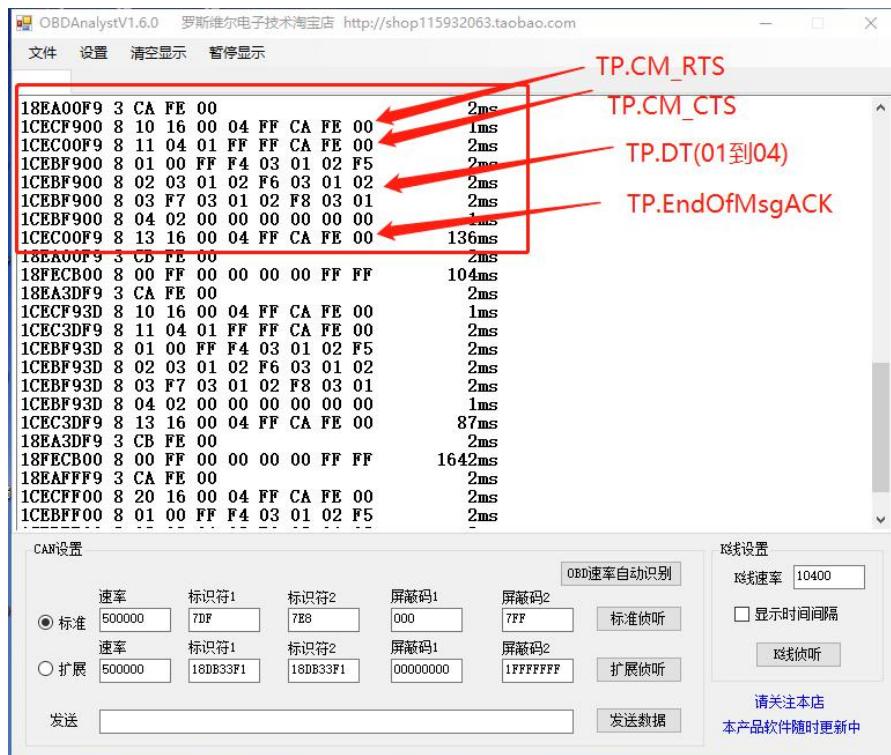
再看数据域，这个对比 TP.CM_BAM 数据域就简单多了，每帧除了第一字节是序号 (Sequence number) 外，2 到 8 字节都是用于承载应用层数据信息的，所以关于上图中 2 到 8 字节关于故障码信息的编码，我们会在接下来的关于应用层定义再具体解读。细心的朋友发现截图 4 关于 2 到 8 字节定义讲到 用不到的字节需要填充 0xFF。所以上图中最后 6 个字节是 00 需要把它填充为 FF。这将在 HD-OBD 模拟器资料 V1.0.9 进行修正。

多帧传输 RTS/CTS 模式：

依然设置 HD-OBD 模拟器故障码，如下图所示：

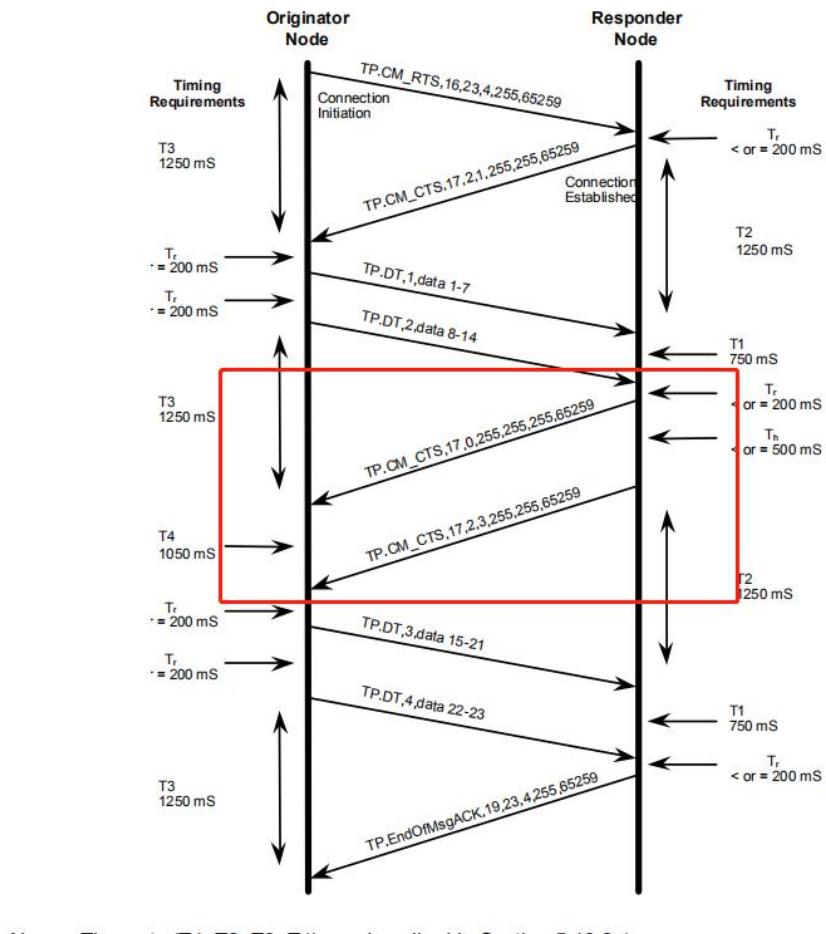


相同办法利用 OBD 分析仪采集 Neulen C300 开发板和 HD-OBD 模拟器通信的数据，如下图所示。



Neulen C300 开发板发送请求命令 (18EA00F9 3 CA FE 00) 后，HD-OBD 模拟器响应 TP.CM_CTS，告诉 Neulen C300 开发板即将要传送的数据包基本信息。Neulen C300 开发板准备好后发送 TP.CM_CTS 命令给模拟器确认可接收 TP.DT 数据，随后模拟器响应 TP.DT 数据。Neulen C300

开发板接收模拟器响应的 TP.DT 数据完毕后, 最后 Neulen C300 开发板 发送 TP.EndOfMsgACK 给模拟器 确认信息传送结束。整个传输流程可以参考下面截图 5, 它是 SAEJ1939-21 数据传输流程图。



(截图 5)

这里我圈出来的红色方框处多了两个 TP.CM_CTS 命令, 这个代表数据较多的情况下, 如果您诊断端的 CPU 处理不过来, 可以允许数据分多次进行传输, 待 CPU 处理完毕后发送 TP.CM_CTS 命令继续传输。这种情况我们比较少遇到, 当前一般的诊断数据我们用 STM32 都可以一次性接收处理完毕, 51 单片机可能需要这种形式的传输。

另外看下面截图 6, 这是传输过程出现错误的处理办法, 我们的 Neulen C300 开发板采用的是 STM32 单片机, 这种错误并不会出现, 主要出现在频率较低的 8 位单片机。具体看下截图 6.

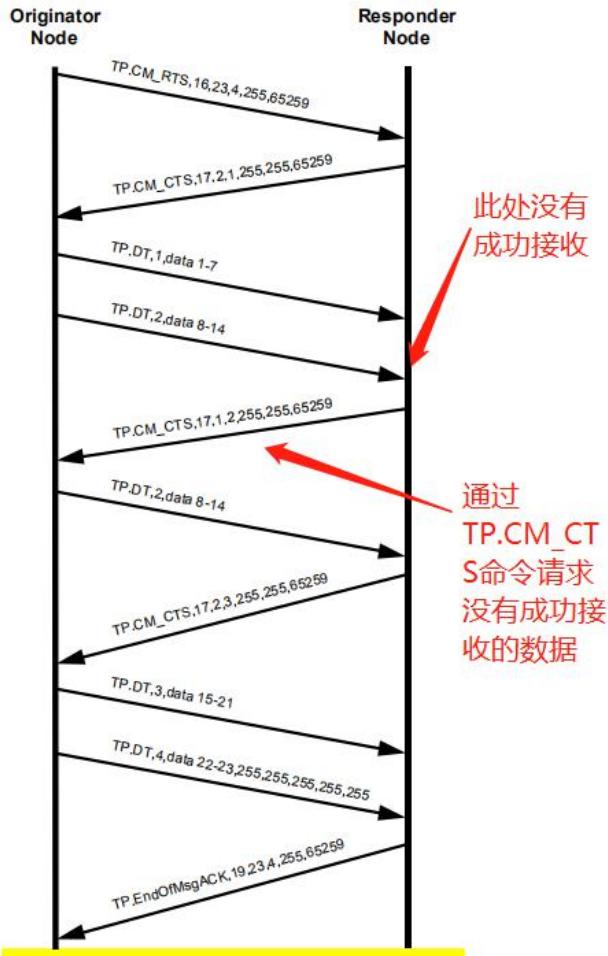


FIGURE C2 DATA TRANSFER WITH ERRORS

(截图 6)

当 TP.DT 在传输过程中出现丢帧的情况，接收端可以通过 TP.CM_CTS 再次请求响应对应丢失的数据。如上图所示，TP.DT2 由于接收端 CPU 没有成功处理，可以通过 TP.CM_CTS 再次请求 TP.DT2。

我们已经了解了 RTS/CTS 模式的传输，以下我们详细看下 TP.CM_RTS, TP.CM_CTS 和 TP.EndOfMsgACK 数据的具体定义。其中还有一个 TP.DT 数据在 CM_BAM 模式里做了介绍，这里就不再重复了。首先明确一点无论是 TP.CM_BAM,TP.CM_RTS,TP.CM_CTS 和 TP.EndOfMsgACK 它们的 PGN 是一样的，优先级，PF,PS 都是一致的，所以它们的 CAN 标识符遵守“截图 1”的定义，但是 TP.CM_RTS,TP.CM_CTS 和 TP.EndOfMsgACK 目标地址必须是特定目标地址，当前数据目标地址是 F9。源地址是 00. 所以当前 CAN 标识符被定义为 0x1CECF900。这些数据主要区别在数据域。

OBD 分析仪采集到的 TP.CM_RTS 数据 1CECF900 8 10 16 00 04 FF CA FE 00

Connection Mode Request to Send (TP.CM_RTS): Destination Specific	
Byte:	1
	Control byte = 16, Destination Specific Request_To_Send (RTS)
	2,3
	Total message size, number of bytes
	4
	Total number of packets
	5
	Maximum number of packets that can be sent in response to one CTS.
	FF ₁₆ indicates that no limit exists for the originator.
	6-8
	Byte 6
	Parameter Group Number of the packeted message
	Byte 7
	Parameter Group Number of requested information
	(8 LSB of parameter group number, bit 8 most significant)
	Byte 8
	Parameter Group Number of requested information
	(2nd byte of parameter group number, bit 8 most significant)
	(8 MSBs of parameter group number, bit 8 most significant)

10 占用第一个字节， 表示 TP.CM_RTS 控制字节。

16 00 占用第二字节， 第三字节， 表示信息长度。

04 占用第四字节， 表示 TP.DT 数据包数。

FF 占用第五字节， 表示响应一个 CTS 最大能响应多少个数据包， 设置为 FF 表示没有限制。

CA FE 00 占用第六字节， 第七字节， 第八字节， 表示 PGN。

OBD 分析仪采集到的 TP.CM_RTS 数据 1CECF900 8 11 04 01 FF FF CA FE 00

Connection Mode Clear to Send (TP.CM_CTS): Destination Specific	
Byte:	1
	Control byte = 17, Destination Specific Clear_To_Send (CTS)
	2
	Number of packets that can be sent. This value shall be no larger than the value in byte 5 of the RTS message.
	3
	Next packet number to be sent
	4-5
	Reserved for assignment by SAE, these bytes should be filled with FF ₁₆
	6-8
	Parameter Group Number of the packeted message

11 占用第一字节， 表示 TP.CM_CTS 控制字节。

04 占用第二字节， 表示响应发送的数据包数目， 我们看到这里是 4 就代表 故障码由 4 个 TP.DT 数据包组成， 同时要求 这个数值不能大于 TP.CM_RTS 值。但是我们的 TP.CM_RTS 设置为 FF， 所以该值没有限制。

01 占用第三字节， 表示响应发送的数据包从哪个数据包开始发送， 这里是 1 就代表响应发动的 TP.DT 数据,它的序列号是从 1 开始。

FF FF 占用第四字节和第五字节， 是预留 SAE 定义用， 填充 FF 即可

CA FE 00 占用第六字节， 第七字节， 第八字节， 表示 PGN。

OBD 分析仪采集到的 TP.CM_EndOfMsgACK 数据 1CECF900 8 13 16 00 04 FF CA FE 00

End of Message Acknowledgment (TP.CM_EndOfMsgACK): Destination Specific
 Byte: 1 Control byte = 19, End_of_Message Acknowledge
 2,3 Total message size, number of bytes
 4 Total number of packets
 5 Reserved for assignment by SAE, this byte should be filled with FF₁₆
 6-8 Parameter Group Number of the packeted message

13 占用第一字节，表示 TP.CM_EndOfMsgACK 控制字节。

16 00 占用第二字节，第三字节，表示接收到的数据长度。

04 占用第四字节，表示接收到数据包的数量。

FF 占用第五字节，表示 SAE 定义预留，填充 FF。

CA FE 00 占用第六字节，第七字节，第八字节，表示 PGN。

OBD 诊断协议的信息通信管理还有一点是时间超时问题，对于物理层基于 CAN 通信的 OBD 协议来说，编写诊断程序的时候，这一点显得不是那么重要和严谨。我们看截图 3 和截图 5，在下面再次插入着两个图。

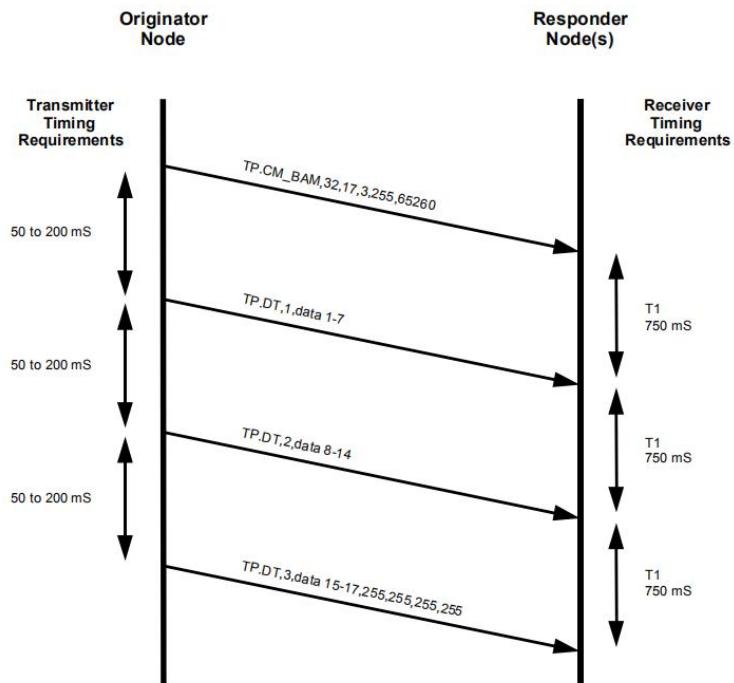
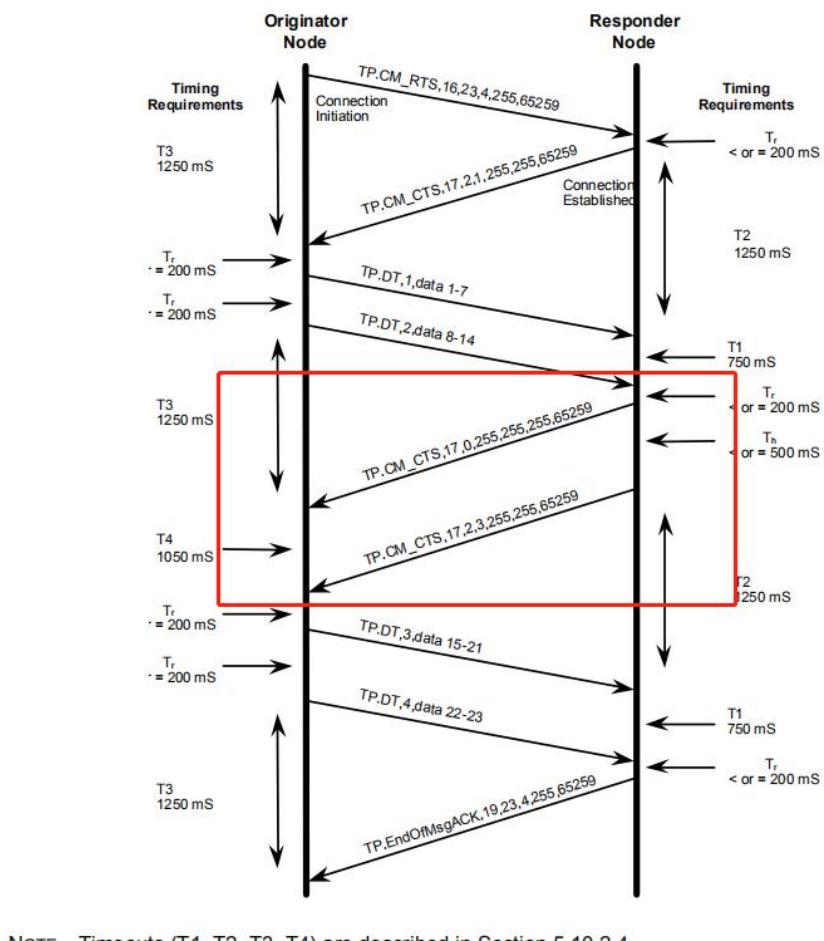


FIGURE C3 BROADCAST DATA TRANSFER



在数据请求和响应之间，有一个时间间隔，这个时间间隔在截图中已经标注了，或者是 1250 毫秒，200 毫秒，750 毫秒等等数值，我们编写程序要充分考虑这个时间问题，要在规定的时间内做出对应的响应，否则需要重新进行数据的请求。而对于诊断程序来说，请求和响应数据之间大于要求的时间即可，大于多少并不是那么严谨，可根据客户对产品的直观感受进行调整。

4. 应用层如何定义

对于汽修诊断设备，常用的诊断功能无非就是读故障码，数据流和版本信息这三大功能。所以我们的 C300 开发板因为要求通用性以便兼容大部分的品牌车型。C300 开发板第一版本软件也是围绕这三大功能进行开发的。当然后续版本我们会推出一些针对特定车型的功能，这部分协议应用层将不是我们从网络所能搜索到的。本节所讲的应用层协议均出自 SAEJ1939-7X。X 是个任意数值，因为不同的功能其后缀编号会有所不同，但是 SAEJ1939 的应用层均定义在后缀编号以 7 开头的文档中。

解读 SAEJ1939 协议，首先了解一个缩写 SPN，全称 Suspect Parameter Number 译为可疑参数编号。通过这个编号可以找到编号对应的数据名称，比如发动机转速，冷却液温度等。它是一个 19bit 的编号，但是这个编号大多只存在协议文档中作为一个基本索引，并不体现在实际数据中，除了故障码。具体定义可以在 SAEJ1939 协议的 Appendix C 列表中找到。

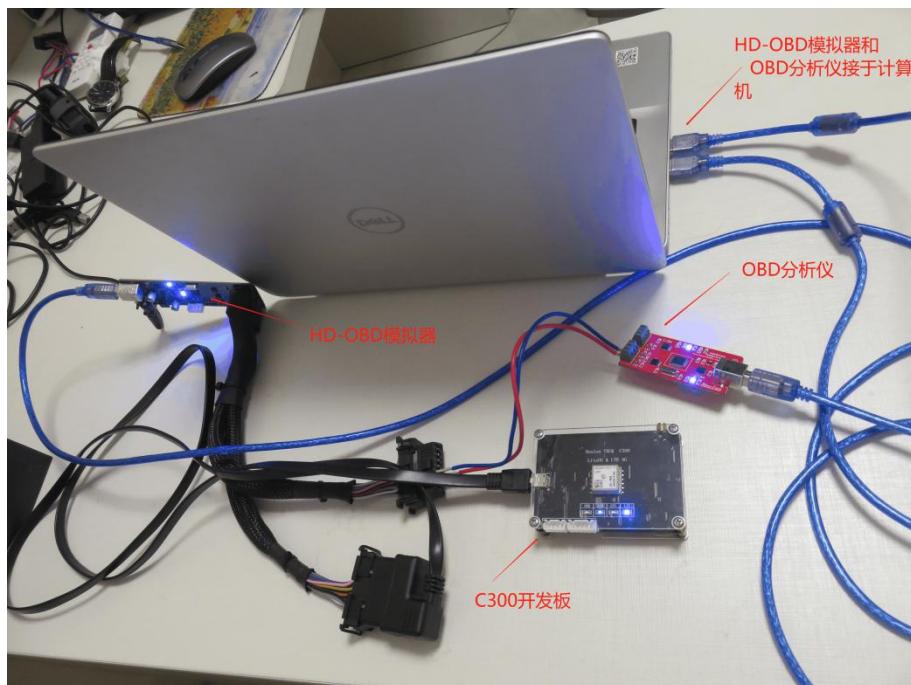
3.2.7 Suspect Parameter Number (SPN)

A Suspect Parameter Number (SPN) is a 19 bit number used to identify a particular element, component, or parameter associated with an ECU. This capability is especially useful for diagnostics, permitting an ECU which has detected a fault associated with a particular component, such as a sensor, to transmit a fault message identifying the faulty component. SPNs are assigned by the Committee and are listed in Appendix C. The first 511 SPNs are reserved and will be assigned, when possible, to the exact same number as the Parameter Identifier (PID) of J1587. For example, J1587 PID 91 is "Percent Accelerator Pedal Position" and an accelerator pedal position parameter fault could be reported in J1939 by using SPN 91. All following SPNs will be assigned in order as they are received.

Due to the very large number of SPNs which may ultimately be assigned, and their assignment in order of request, it will be very difficult for one interested in finding the SPN value of a particular component of interest simply by looking through the table. To facilitate the verification that new SPN requests are not duplications of existing assignments, the committee retains this table as an MS Excel™ spreadsheet, with additional data beyond that shown in Table C1. This permits sorting based upon SPN number, name, description, attribute (actuator, pressure, temperature, solenoid, etc.), J1587 attributes (MID, PID, SID), J1939 document paragraph, source name, and source address. It would be desirable for those developing J1939 applications or wishing to request the assignment of a new SPN to have access to an up-to-date version of this spreadsheet so that they can perform various sorts and searches of the data. At the time of publication, the SAE has not yet determined how this data can best be made available to the users of J1939 who are not committee participants.

读故障码

读故障码可参考SAEJ1939-73 协议。在协议中有读当前激活故障码（Active Diagnostic Trouble Codes），历史故障码（Previously Active Diagnostic Trouble Codes）。先看读当前激活故障码。依然采用 Neulen C300 开发板和 HDOBD 模拟器通信，用 OBD 分析仪采集数据进行分析。这样通过数据解读协议更明了。



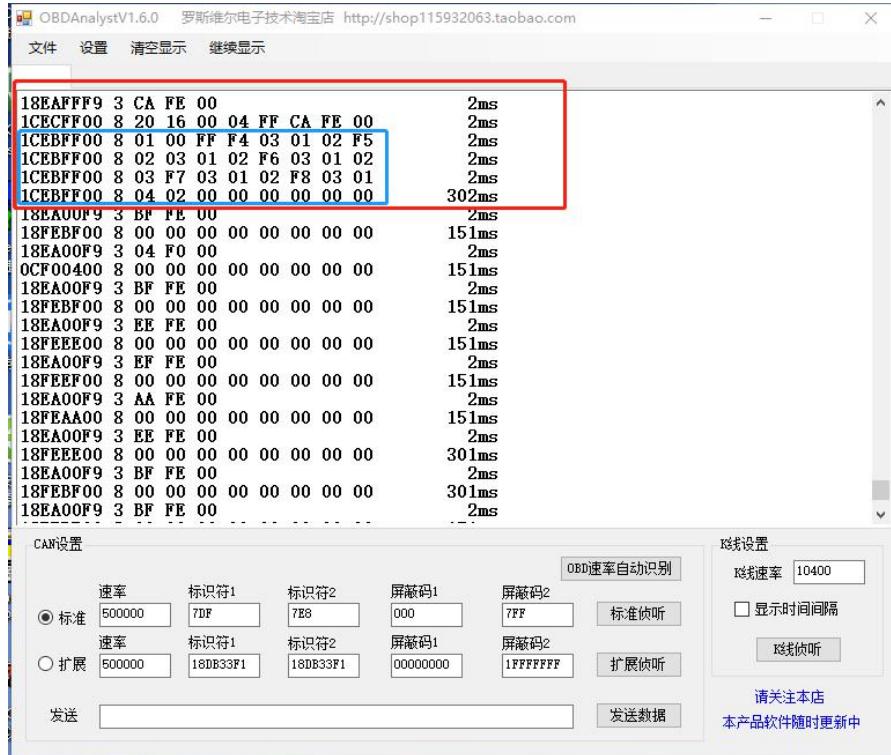
(接线图)



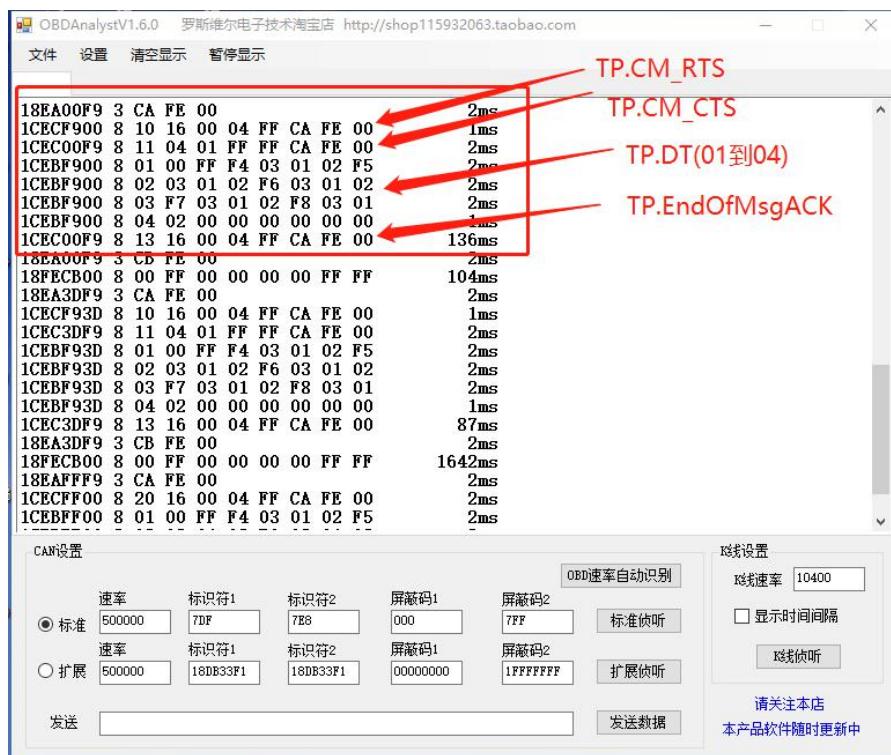
(设置故障码:SPN1012-1-2;SPN1013-1-2;SPN1014-1-2;SPN1015-1-2;SPN1016-1-2)



(C300 开发板读取到的故障码)



(采集 C300 开发板获取故障码过程，模拟器以 TP.CM_BAM 模式响应故障码)



(采集 C300 开发板获取故障码过程，模拟器以 TP.CM_RTS/CTS 模式响应故障码)

以上采集到的故障码响应数据，不管是 BAM 模式还是 RTS/CTS 模式，去除数据链路层定义的通信协议传输模式规则，提炼其承载的应用层数据如下所示。

00 FF F4 03 01 02 F5 03 01 02 F6 03 01 02 F7 03 01 02 F8 03 01 02

这个数据其实是响应的 TP.DT 数据的数据域去除第一字节的序列号后获得的。那么用这个数据对照下面从 SAEJ1939-73 应用层协议截图来解释每个字节的含义。

5.7.1 ACTIVE DIAGNOSTIC TROUBLE CODES (DM1)—The information communicated is limited to the currently active diagnostic trouble codes preceded by the diagnostic lamp status. Both are used to notify other components on the network of the diagnostic condition of the transmitting electronic component. The data contains the lamp status and a list of diagnostic codes and occurrence counts for currently active diagnostic trouble codes. This is all DTCs including those that are emissions related.

The currently defined lamps (Malfunction Indicator Lamp, Red Stop Lamp, Amber Warning Lamp, and Protect Lamp) are associated with DTCs. If the transmitting electronic component does not have active DTCs, then the lamp status from that component will indicate that the lamps should be off. However, the component controlling the actual lamp illumination must consider the status from all components that use these lamps before changing the state of the lamps.

There may be applications that require additional lamp definitions to accomplish their function (e.g. a lamp that indicates when cruise control is actively controlling would require a separate lamp in another PG).

Transmission Rate:	A DM1 message is transmitted whenever a DTC becomes an active fault and at a normal update rate of only once per second thereafter. If a fault has been active for 1 second or longer, and then becomes inactive, a DM1 message shall be transmitted to reflect this state change. If a different DTC changes state within the 1 second update period, a new DM1 message is transmitted to reflect this new DTC. To prevent a high message rate due to intermittent faults that have a very high frequency, it is recommended that no more than one state change per DTC per second be transmitted. Thus a DTC that becomes active/inactive twice within a 1 second interval, such as shown in Example Case 1, would have one message identifying the DTC becoming active, and one at the next periodic transmission identifying it being inactive. This message is sent only when there is an active DTC existing or in response to a request. Note that this Parameter Group will require using the "multipacket Transport" Parameter Group (reference SAE J1939-21) when more than one active DTC exists.		
Data Length:	Variable		
Data page:	0		
PDU Format:	254		
PDU Specific:	202		
Default Priority:	6		
Parameter Group Number:	65226 (00FECA ₁₆)		
Byte: 1	bits 8-7	Malfunction Indicator Lamp Status	See 5.7.1.1
	bits 6-5	Red Stop Lamp Status	See 5.7.1.2
	bits 4-3	Amber Warning Lamp Status	See 5.7.1.3
	bits 2-1	Protect Lamp Status	See 5.7.1.4
Byte: 2	bits 8-7	Reserved for SAE assignment Lamp Status	
	bits 6-5	Reserved for SAE assignment Lamp Status	
	bits 4-3	Reserved for SAE assignment Lamp Status	
	bits 2-1	Reserved for SAE assignment Lamp Status	
Byte: 3	bits 8-1	SPN, 8 least significant bits of SPN (most significant at bit 8)	See 5.7.1.5
Byte: 4	bits 8-1	SPN, second byte of SPN (most significant at bit 8)	See 5.7.1.5
Byte: 5	bits 8-6	SPN, 3 most significant bits (most significant at bit 8)	See 5.7.1.5
	bits 5-1	FMI (most significant at bit 5)	See 5.7.1.6
Byte: 6	bit 8	SPN Conversion Method	See 5.7.1.7
	bits 7-1	Occurrence Count	See 5.7.1.8

截图中提供了完整的当前激活故障码响应信息格式，数据长度，数据页，PDU 格式，PS，优先级,PGN 都给出了，这些如何组成一个完整的响应数据，大家可以看前面的信息通信管理的内容，这里我们只看数据域定义和上面提炼出来的数据一一对应入座。

00 第一字节，表示故障灯状态。显然这个字节在我们 HDOBD 模拟器里还不能模拟，默认 00，后期 HDOBD 模拟器会增加这块模拟内容。

FF 第二字节，表示预留的故障灯状态定义字节。因为是预留，我们也不能自行定义，所以这个按照官方要求填充 FF。

F4 03 01 02 第三字节，第四字节，第五字节，第六字节四个字节，表示故障码编号。下图所示更直观解析这 4 个字节如何表示一个故障码的。

Table 2: DTC Representation in CAN Data Frame for DM1 (Byte 3 Closer to CAN Identifier)

J1939 Frame Format	DTC																												
	Byte 3 8 least significant bits of SPN (bit 8 most significant)								Byte 4 second byte of SPN (bit 8 most significant)				Byte 5 3 most significant bits of SPN and the FMI (bit 8 SPN msb and bit 5 FMI msb)				Byte 6												
	SPN								FMI				CM	OC															
	8	7	6	5	4	3	2	1	8	7	6	5	4	3	2	1	8	7	6	5	4	3	2	1					
	1	0	1	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	0	0	0	1	0	1	0

其中第三字节,第四字节和第五字节的高三位定义为 SPN 值,第五字节的低五位表示 FMI. FMI 定义的是故障模式，说明具体故障原因，其值定义列表在 SAEJ1939-73 协议的 Appendix A. 第六字节第八位 CM 表示 SPN 转换办法，也就是这四个字节前 19 位如何表示 SPN 的。当该值为 1 的时候表示采用版本 1 到版本 3 的转换办法，这里稍微提及。目前主要采用版本 4，也就是说 CM=0 的时候表示采用 SPN 转换办法是版本 4。版本 4 就是 19 位 SPN 值低字节在总线上先传输，高字节最后传输，就是上图所示的内容。低字节放在 Byte3, 上图标题括号尤其重点讲到 Byte3 Closer to CAN Identifier. 第三字节更靠近 CAN 标识符。也就是第三字节优先在总线上传输。第六字节的最低七位 OC 表示故障出现的次数，这是故障发生计数器。所以在这组应用层数据里接下来的 第七字节，第八字节，第九字节，第十字节组成的第二个故障码数据也和上面解析一致。下面罗列出每个故障码并最终解释它们实际含义。

第一个故障码数据 **F4 03 01 02**

SPN = 0x03f4 = 1012

FMI = 1

OC = 2

显示故障码 SPN1012-1-2

故障解释：Trip Drive Fuel Economy (Gaseous) —DATA VALID BUT BELOW NORMAL OPERATIONAL RANGE - MOST SEVERE LEVEL 故障发生 2 次。

这个故障解释如何查找，上面故障码定义内容我已提及，这里结合实际故障码再次具体告诉大家怎么找。

故障点 Trip Drive Fuel Economy (Gaseous)是通过 SPN=1012 查找 SAEJ1939 协议的 Appendix C 获得，如下图所示。

SAE J1939 Revised JAN20					
Rev	SPN	SPN Name	J1939 Reference		
			J1939 Doc	PGN Number	Pos in PG
	1012	Trip Drive Fuel Economy (Gaseous)	-71	65208 21-22	
	1013	Trip Maximum Engine Speed	-71	65207 1-2	
	1014	Trip Average Engine Speed	-71	65207 3-4	
	1015	Trip Drive Average Load Factor	-71	65207 5	
	1016	Total Drive Average Load Factor	-71	65207 6	
	1017	Total Engine Cruise Time	-71	65207 7-10	
	1018	Trip Maximum Vehicle Speed	-71	65206 1-2	
	1019	Trip Cruise Distance	-71	65206 3-6	
	1020	Trip Number of Hot Shutdowns	-71	65205 1-2	
	1021	Trip Number of Idle Shutdowns	-71	65205 3-4	
	1022	Trip Number of Idle Shutdown Overrides	-71	65205 5-6	
	1023	Trip Sudden Decelerations	-71	65205 7-8	

- 110 -

故障内容 DATA VALID BUT BELOW NORMAL OPERATIONAL RANGE - MOST SEVERE LEVEL 通过 FMI=1 在 SAEJ1939-73 协议的 Appendix A 中获得，如下图所示。

A.1.2 FMI and Description

A.1.2.1 FMI=0—DATA VALID BUT ABOVE NORMAL OPERATIONAL RANGE - MOST SEVERE LEVEL

The signal communicating information is within a defined acceptable and valid range, but the real world condition is above what would be considered normal as determined by the predefined most severe level limits for that particular measure of the real world condition (*Region e* of the signal range definition). Broadcast of data values is continued as normal.

A.1.2.2 FMI=1—DATA VALID BUT BELOW NORMAL OPERATIONAL RANGE - MOST SEVERE LEVEL

The signal communicating information is within a defined acceptable and valid range, but the real world condition is below what would be considered normal as determined by the predefined least severe level limits for that particular measure of the real world condition (*Region d* of signal range definition). Broadcast of data values is continued as normal.

第二个故障码数据 F5 03 01 02

SPN = 0x03f5 = 1013

FMI = 1

OC = 2

显示故障码 SPN1013-1-2

故障解释：Trip Maximum Engine Speed —DATA VALID BUT BELOW NORMAL OPERATIONAL RANGE - MOST SEVERE LEVEL 故障发生 2 次。

第三个故障码数据 F6 03 01 02

SPN = 0x03f6 = 1014

FMI = 1

OC = 2

显示故障码 SPN1014-1-2

故障解释: Trip Average Engine Speed —DATA VALID BUT BELOW NORMAL OPERATIONAL RANGE - MOST SEVERE LEVEL 故障发生 2 次

第四个故障码数据 F7 03 01 02

SPN = 0x03f7 = 1015

FMI = 1

OC = 2

显示故障码 SPN1015-1-2

故障解释: Trip Drive Average Load Factor —DATA VALID BUT BELOW NORMAL OPERATIONAL RANGE - MOST SEVERE LEVEL 故障发生 2 次

第三个故障码数据 F8 03 01 02

SPN = 0x03f6 = 1016

FMI = 1

OC = 2

显示故障码 SPN1016-1-2

故障解释: Total Drive Average Load Factor —DATA VALID BUT BELOW NORMAL OPERATIONAL RANGE - MOST SEVERE LEVEL 故障发生 2 次

下面我们看下 X431 对这些故障码的解释是否基本一致.





故障解释，X431 中文解释和我们查找到的英文解释是一致的。

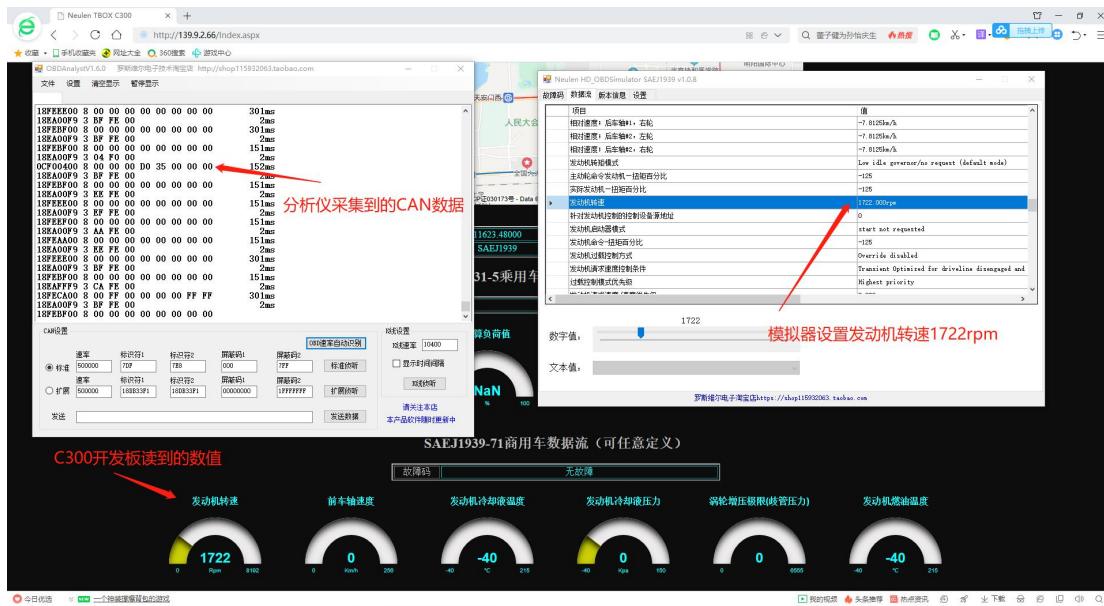
这是 PGN=0XFeca 的当前激活故障码的协议解读，除此之外还有历史故障码等，其读码过程和故障解释都是一样的，只需改变它们的请求 PGN 就可以了，如下图所示是历史故障码的定义截图。（PGN=0XFecb）

- 5.7.2 PREVIOUSLY ACTIVE DIAGNOSTIC TROUBLE CODES (DM2)—The information communicated is limited to the previously active trouble codes. It is used to notify other components on the network of the diagnostic condition of the transmitting electronic component. The data contains a list of diagnostic codes and occurrence counts for previously active trouble codes. Whenever this message is sent, it should contain a previously active trouble codes with an occurrence count not equal to zero. Note that this Parameter Group will be sent using the "Multipacket Transport" Parameter Group as specified in SAE J1939-21 where applicable.

Transmission Rate:	On request using PGN 59904 A NACK is required if PG is not supported (see SAE J1939-21 PGN 59392)	See SAE J1939-21
Data Length:	Variable	
Data page:	0	
PDU Format:	254	
PDU Specific:	203	
Default Priority:	6	
Parameter Group Number:	65227 (00FECB ₁₆)	
Byte: 1	bits 8-7 Malfunction Indicator Lamp Status bits 6-5 Red Stop Lamp Status bits 4-3 Amber Warning Lamp Status bits 2-1 Protect Lamp Status	See 5.7.1.1 See 5.7.1.2 See 5.7.1.3 See 5.7.1.4
Byte: 2	bits 8-7 Reserved for SAE assignment Lamp Status bits 6-5 Reserved for SAE assignment Lamp Status bits 4-3 Reserved for SAE assignment Lamp Status bits 2-1 Reserved for SAE assignment Lamp Status	
Byte: 3	bits 8-1 SPN, 8 least significant bits of SPN (most significant at bit 8)	See 5.7.1.5
Byte: 4	bits 8-1 SPN, second byte of SPN (most significant at bit 8)	See 5.7.1.5
Byte: 5	bits 8-6 SPN, 3 most significant bits (most significant at bit 8) bits 5-1 FMI (most significant at bit 5)	See 5.7.1.5 See 5.7.1.6
Byte: 6	bit 8 SPN Conversion Method bits 7-1 Occurrence Count	See 5.7.1.7 See 5.7.1.8

读数据流

数据流应用层协议在 SAEJ1939-71 协议中具体定义。关于数据流的请求命令依然是参照 SAEJ1939-21 协议关于请求的相关格式执行，具体可以回顾我们本节前面的内容。CAN 标识符使用 18EA00F9，这里通常使用特殊目标地址，00 是发动机控制单元。DLC 固定是 3。数据域就是 3 个字节的 PGN 了。为了更好的理解和解读数据流协议内容，我们做如下实验采集数据。



HDOBD 模拟器任意设置一个数据流参数值，这里我们以发动机转速为例子，这里设置任意一个值为 1722 转，此时可以看到 C300 开发板读取到的发动机转速数据流的值为 1722 转。主要看分析仪采集到的原始数据对照协议如何转换的。此时发现以 0CF00400 为标识符的数据发生变化，它的请求命令是 18EA00F9 3 04 F0 00。把数据摘录如下所示。

发动机转速请求命令 18EA00F9 3 04 F0 00

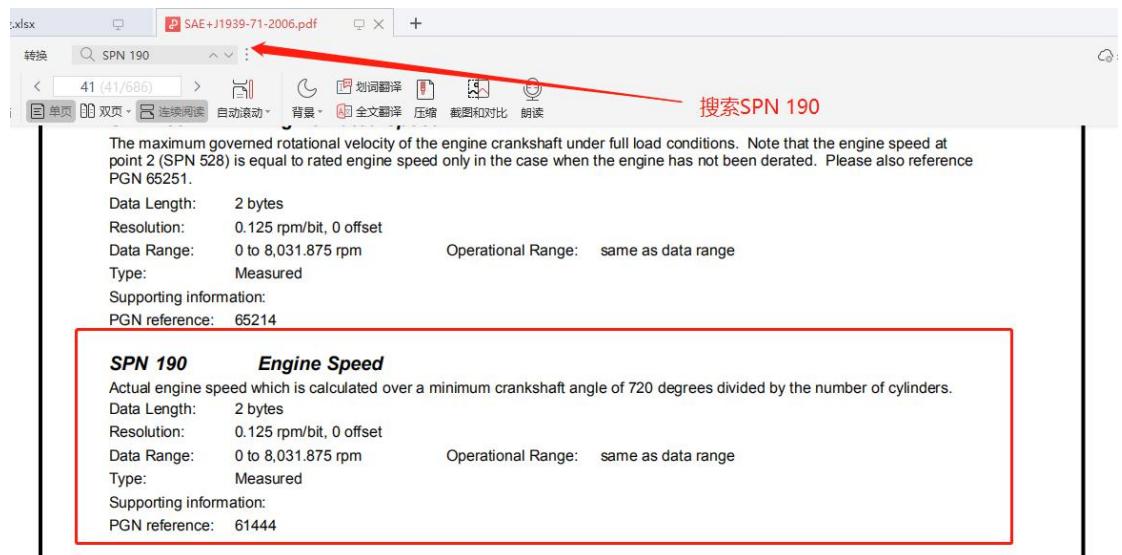
发动机转速响应命令 0CF00400 8 00 00 00 D0 35 00 00

关于请求命令格式我们这里不多做解释，前面内容已经详细讲解过了。通过请求命令和响应数据我们可以知道发动机转速的 PGN 是 0xF004(十进制 61444)。由于拖动 HDOBD 模拟器进度条设置转速值的时候发现当前 D0 35 这个位置的数据发生变化，由此也可以断定第四字节和第五字节是表示发动机转速的。那么根据以上信息我们查下 SAEJ1939-71 协议发动机转速的具体定义，通过这个查询的办法让大家学会解读 SAEJ1939 的数据流协议。

通过 PGN=61444 搜索 SAEJ1939-71 协议里的内容，可快速定位到该 PGN 的定义，如下图所示。

Start Position	Length	Parameter Name	SPN
1.1	4 bits	Engine Torque Mode	899
2	1 byte	Driver's Demand Engine - Percent Torque	512
3	1 byte	Actual Engine - Percent Torque	513
4-5	2 bytes	Engine Speed	190
6	1 byte	Source Address of Controlling Device for Engine Control	1483
7.1	4 bits	Engine Starter Mode	1675
8	1 byte	Engine Demand - Percent Torque	2432

从协议截图中清晰注明了，承载发动机转速的响应数据的数据长度 8 个字节，PF=240, PS=4, 优先级 3。那么组成的 CAN 标识符就是 0CF00400. 在数据域部分，也明确定义了第四字节和第五字节长度 2 个字节的数据为发动机转速，所以数据中的 D0 35 正是发动机转速的原始值。那么原始值 D0 35 也就是 0x35D0 和 1722 之间是如何一个转换关系呢？请看后面标注的可疑参数编号 SPN=190. 我们依然使用文档搜索 SPN 190 快速定位到这个 SPN 的解决办法。如下图所示。



这里明确定义了发动机转速由原始数据转换成应用数据的关系。数据长度 2 个字节；解决算法 0.125 每比特，起始 0 开始；数据范围 0 到 8031.875 rpm. 好了，这么多信息足以计算。0x35D0 十进制是 13776. $13776 * 0.125 = 1722$. 到此大家应该了解整个数据流的定义和转换过程了。

7.1.2 SAEJ1939 协议在 Neulen TBOX C300 开发板中的程序实现

以上内容我们比较全面分析了 SAEJ1939 各层协议的定义内容，当然我们是挑选对实现产品有用的内容进行讲解。其它内容对实现产品没有任何作用我们就不细致讲解。下面我们继续看下如何通过代码来实现这些内容。

很多朋友在编写诊断程序的时候很纠结是否严格按照协议分层结构去编写程序，这跟读死书没什么两样。对于我们的 C300 开发板，本人是通过一个函数来实现 物理层，数据链路层和会话层的。应用层必须进行剥离，这对于二次开发和后续升级很重要。所以真正编写诊断程序可以简单把我们的程序分成两部分，第一部分就是实现物理层，数据链路层和会话层功能的代码。这部分代码为了让第二部分的应用层调用方便，C300 开发板把它集成在了一个函数内。即函数 NL_OBD_SendCANFrame。下面利用 C300 开发板测试这个函数，可以通过测试读当前故障码(DM1)来验证函数对物理层，数据链路层，会话层的处理能力。

NL_OBD_SendCANFrame 函数处理单帧响应。

利用 HD_OBD 模拟器设置当前故障码为 1 个故障码，这时候模拟器对故障码的响应信息只需一个 CAN 帧即可承载故障码信息，即少于 8 个字节的有效信息。



下面是函数 NL_OBD_SendCANFrame 原型，关于它的具体实现办法稍后我们会具体介绍，这里我们先看它的参数都有哪些，它们分别是 协议类型，待发送的 CAN 数据指针，等待响应时间，错误判断四个参数，返回值是汽车响应数据存储地址的指针。所以看出函数通过发送请求数据给汽车或者模拟器来获得汽车或者模拟器响应的数据，响应数据会存储在芯片 RAM 中并返回存储指针。

```

40 /**
41 * @brief OBD接口发送一帧CAN数据
42 * @param pro 协议类型, TxMessage: 待发送CAN数据, Timeout: 等待响应时间 , err: 响应是否成功
43 * @return uint8_t* 类型指针, 指向响应数据。
44 */
45 OBDFlagDef OBDFlag;
46 uint8_t* CAN_RXRAM[200];
47 uint8_t* CAN_PGN[2];
48 uint8_t* NL_OBD_SendCANFrame(ProtocolDef pro, CanTxFrameDef *TxMessage, uint32_t Timeout, NLStatus *err)
49 {
50     uint32_t i;
51     OBDFlag.ProType = pro;
52     if(pro == SAEJ1939)
53     {
54         CAN_PGN[0] = TxMessage->Data[0];
55         CAN_PGN[1] = TxMessage->Data[1];
56     }
57     else if(pro == ISO15765_4STD_500K)
58     {
59         TxMessage->IDE = CAN_ID_STD;
60     }
61     else
62     {
63         TxMessage->IDE = CAN_ID_EXT;
64     }
65     OBDFlag.RxFlag = NL_FAILURE;
66     *err = NL_NOK;
67     _NL_OBD_CAN_Transmit(TxMessage);
68     for (i = 0; i < Timeout; i++)
69     {
70         _NL_Delay(1);
71         if (OBDFlag.RxFlag == NL_SUCCESS)
72         {
73             *err = NL_OK;
74             break;
75         }
76     }
77 }
78 return CAN_RXRAM;
79 }
```

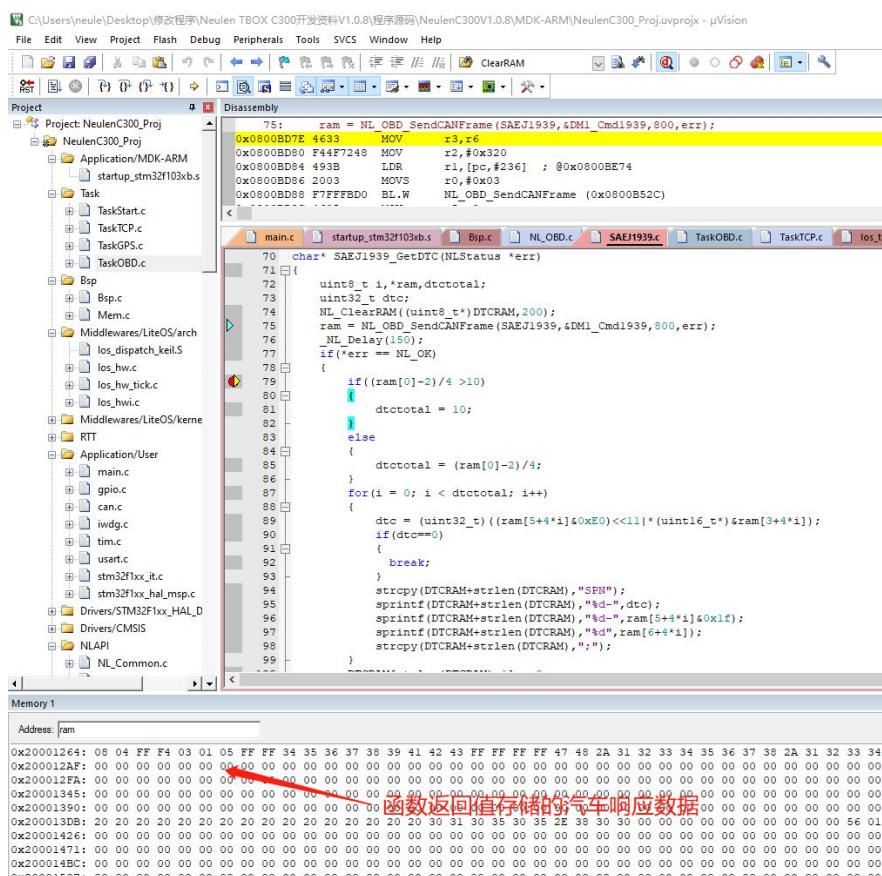
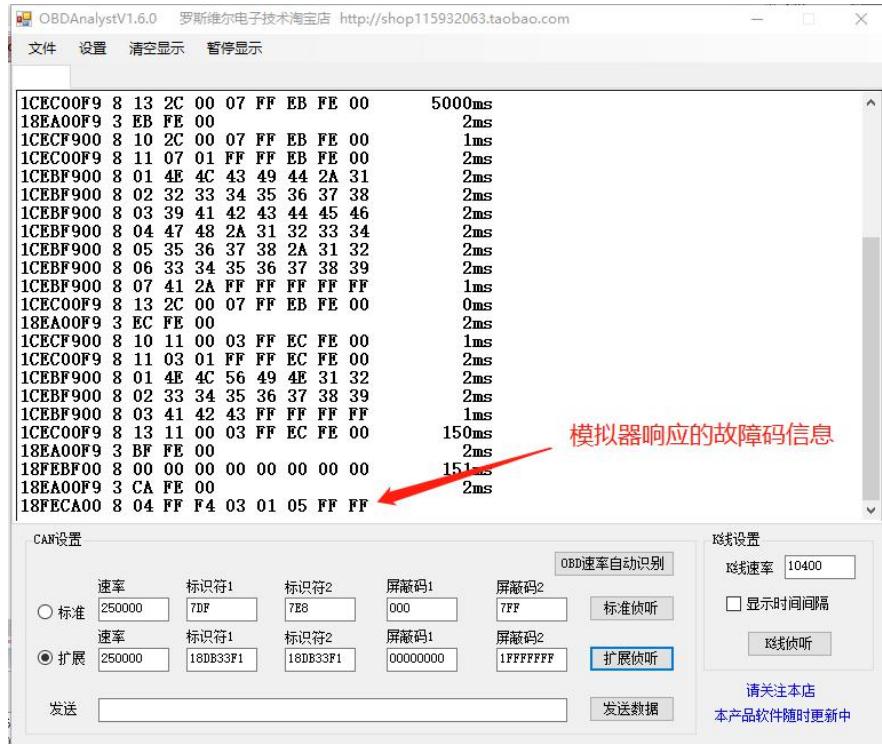
读故障码程序片段如下所示。

```
64 }
65 /* @描述: SAEJ1939读取故障码
66 * @参数: *err :NL_OK:成功 NL_NOK:不成功
67 * @返回值: 故障码存储指针
68 */
69
70 char* SAEJ1939_GetDTC(NLStatus *err)
71 {
72     uint8_t i,*ram,dtcTotal;
73     uint32_t dtc;
74     NL_ClearRAM((uint8_t*)DTCTRAM,200);
75     ram = NL_OBD_SendCANFrame(SAEJ1939,&DM1_Cmd1939,800,err);
76     _NL_Delay(150);
77     if(*err == NL_OK)
78     {
79         if((ram[0]-2)/4 >10)
80         {
81             dtcTotal = 10;
82         }
83         else
84         {
85             dtcTotal = (ram[0]-2)/4;
86         }
87         for(i = 0; i < dtcTotal; i++)
88         {
89             dtc = (uint32_t)((ram[5+4*i]&0xE0)<<11|*(uint16_t*)&ram[3+4*i]);
90             if(dtc==0)
91             {
92                 break;
93             }
94             strcpy(DTCTRAM+strlen(DTCTRAM),"SPN");
95             sprintf(DTCTRAM+strlen(DTCTRAM),"%d-",dtc);
96             sprintf(DTCTRAM+strlen(DTCTRAM),"%d-",ram[5+4*i]&0x1f);
97             sprintf(DTCTRAM+strlen(DTCTRAM),"%d",ram[6+4*i]);
98             strcpy(DTCTRAM+strlen(DTCTRAM),":");
99         }
100        DTCTRAM[strlen(DTCTRAM)-1] = 0;
101    }
102    return DTCTRAM;
103 }
```

程序片段中函数 `NL_OBD_SendCANFrame` 被调用，第一个参数“`SAEJ1939`”表示当前该函数执行的是 `SAEJ1939` 协议，除此之外该函数还可以执行别的 CAN 协议，比如 `ISO15765` 协议，这在下一节内容再具体介绍。第二个参数“`&DM1_Cmd1939`”即 `SAEJ1939-73` 协议下定义的当前故障码。这是请求数据，相关联的定义内容还必须同时参考 `SAEJ1939-21` 协议，如下所示这是 `DM1_Cmd1939` 调用的数据类型，这里值得注意的是当前数据的目标地址被定义为特定目标地址。第三个参数“`800`”等待时间，就是请求数据向汽车或者模拟器发送之后，系统在此将阻塞 800 毫秒的时间，等待汽车或者模拟器的响应，响应与否会直接提现在第四个参数“`err`”上，`err` 是 `NLStatus` 定义变量的指针，指向的变量存储 `NL_OK` 则表示函数发送请求后得到了汽车或者模拟器的正确响应。所以函数 `NL_OBD_SendCANFrame` 被调用之后总在它的后面代码做一个 `if` 判断，判断 `err` 指向的存储单元是否是 `NL_OK`。如果是就可以放心对函数返回值进行应用层处理。

```
4
5 //*****源码由SAEJ1939.C自动生成*****  
6 /*@文件: SAEJ1939.c  
7 *@作者: Tony.Neulen  
8 *@日期: 2019/10/17 10:07  
9 *@描述: SAEJ1939下的函数实现  
10 *@产品购买:https://shop11932063.taobao.com  
11 //*****源码由SAEJ1939.C自动生成*****  
12 #include "includes.h"  
13  
14 //命令  
15 CanTxFrameDef Dm1 Cmd1939 = {0x18EA00F5,CAN_ID_EXT,CAN_RTR_DATA,3,0xCA,0xFE,0x00,0x00,0x00,0x00,0x00,0x00};  
16 CanTxFrameDef VINCmd1939 = {0x18EA00F5,CAN_ID_EXT,CAN_RTR_DATA,3,0xE6,0xFF,0x00,0x00,0x00,0x00,0x00,0x00,0x00};  
17 CanTxFrameDef ReqQmd1939 = {0x18EA00F5,CAN_ID_EXT,CAN_RTR_DATA,3,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};  
18 CanTxFrameDef CTSCmd1939 = {0x1CEC00F5,CAN_ID_EXT,CAN_RTR_DATA,8,0x11,0x00,0x01,0xFF,0xFF,0x00,0x00,0x00,0x00};  
19 CanTxFrameDef ACKACKCmd1939 = {0x1CEC00F5,CAN_ID_EXT,CAN_RTR_DATA,8,0x13,0x00,0x01,0xFF,0xFF,0x00,0x00,0x00,0x00};  
20 //*****源码由SAEJ1939.C自动生成*****  
21 /*@描述: SAEJ1939唤醒判断  
22 *@参数: NONE  
23 *@返回值: NL_OK-支持 NL_NOK-不支持  
24 //*****源码由SAEJ1939.C自动生成*****
```

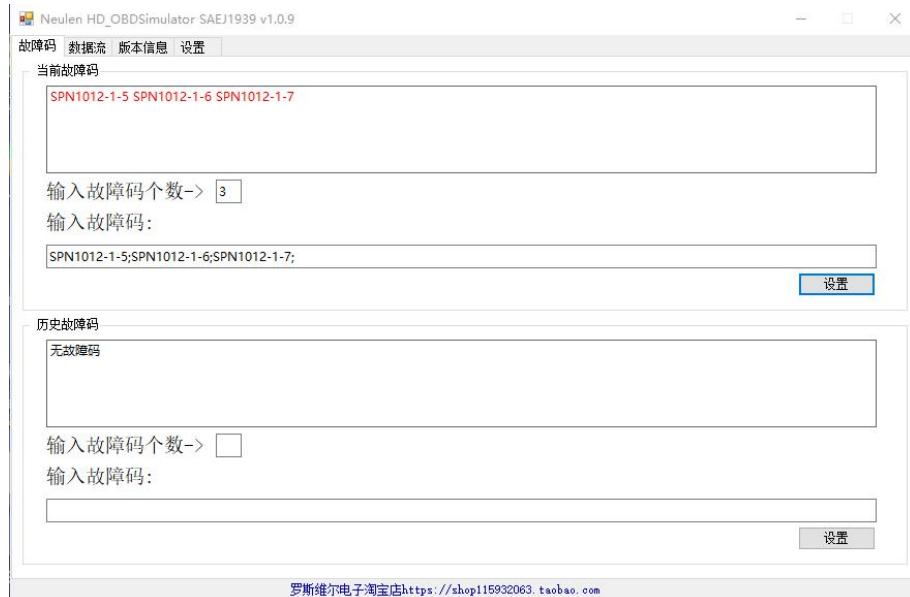
下面我们在 NL_OBD_SendCANFrame 函数后，判断 err 的 if 语句内做一个断点，并在线仿真执行 C300 开发板程序，并利用 OBD 分析仪采集数据做一个对比。如下图所示。



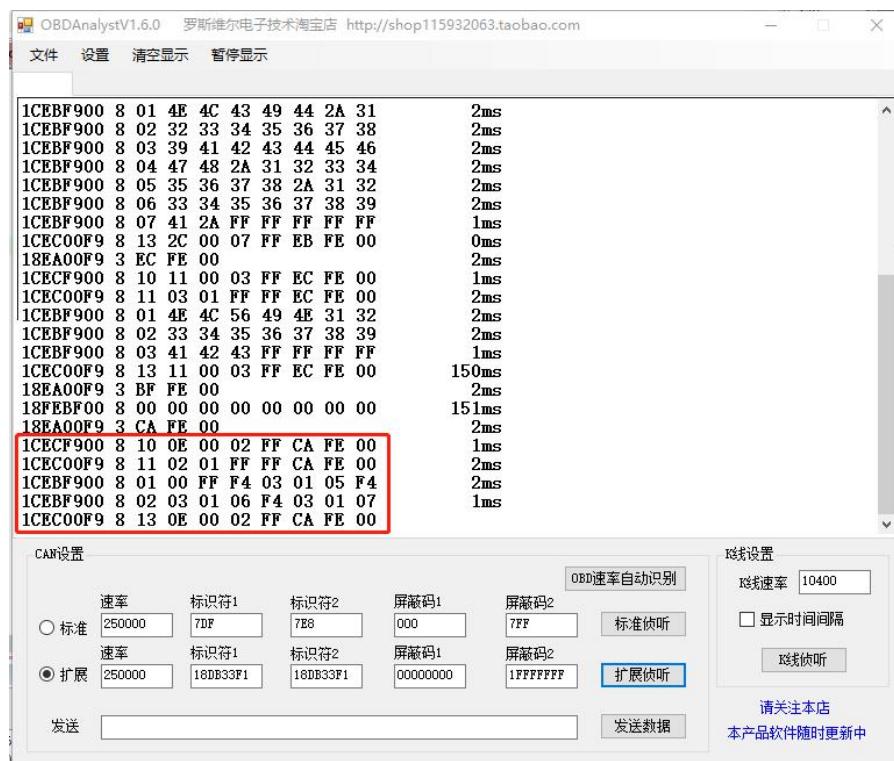
从两张截图中可以看到，模拟器响应的单帧故障码信息被准确存储在了 NL_OBD_SendCANFrame 函数返回值所指向的存储单元中，并且该存储单元结构的第一个字节表示存储有效内容的长度，此时它的值是 8，表示响应数据的数据域 8 个字节被存储。

NL_OBD_SendCANFrame 函数处理 RTS/CTS 模式下的多帧响应。

只要多于 1 个当前故障码，当前故障码的响应信息将会以多帧形式响应。那么为了达到故障码响应多帧的目的，如下图所示设置三个故障码。

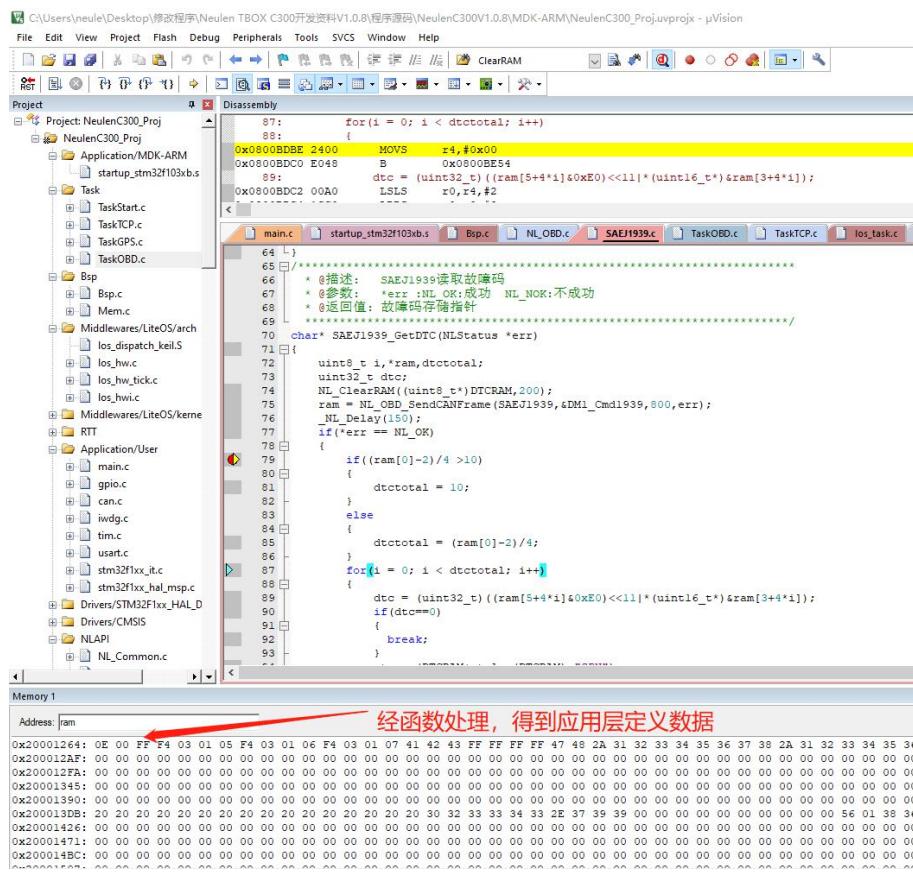


前面已经提及“&DM1_Cmd1939”对应的请求数据，目标地址是特定目标地址的。根据前面小节关于 SAEJ1939 协议的解读我们得知，此种情况下，汽车或者模拟器会以 RTS/CTS 模式进行多帧响应。通信数据如下图所示。



红色方框的数据就是模拟器响应的故障码内容，这是以 RTS/CTS 模式承载的故障码信息。

Neulen 工作室 <https://shop115932063.taobao.com>

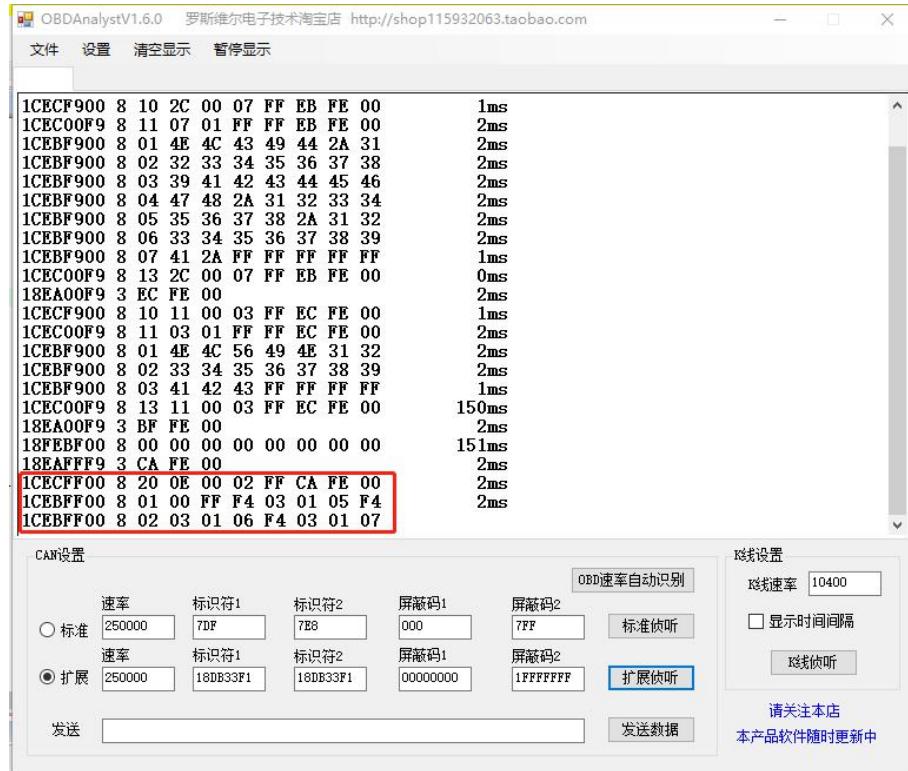


而通过函数 NL_OBD_SendCANFrame 返回值可以看到 SAEJ1939-73 应用层待处理的格式，当然第一字节是我加进去的表示有效字节长度，从第二字节开始 00 表示故障灯状态，第三字节 FF 是预留故障灯状态，第四字节开始，每四个字节表示一个故障码。这样的数据格式就可以完全按照应用层协议进行处理了。

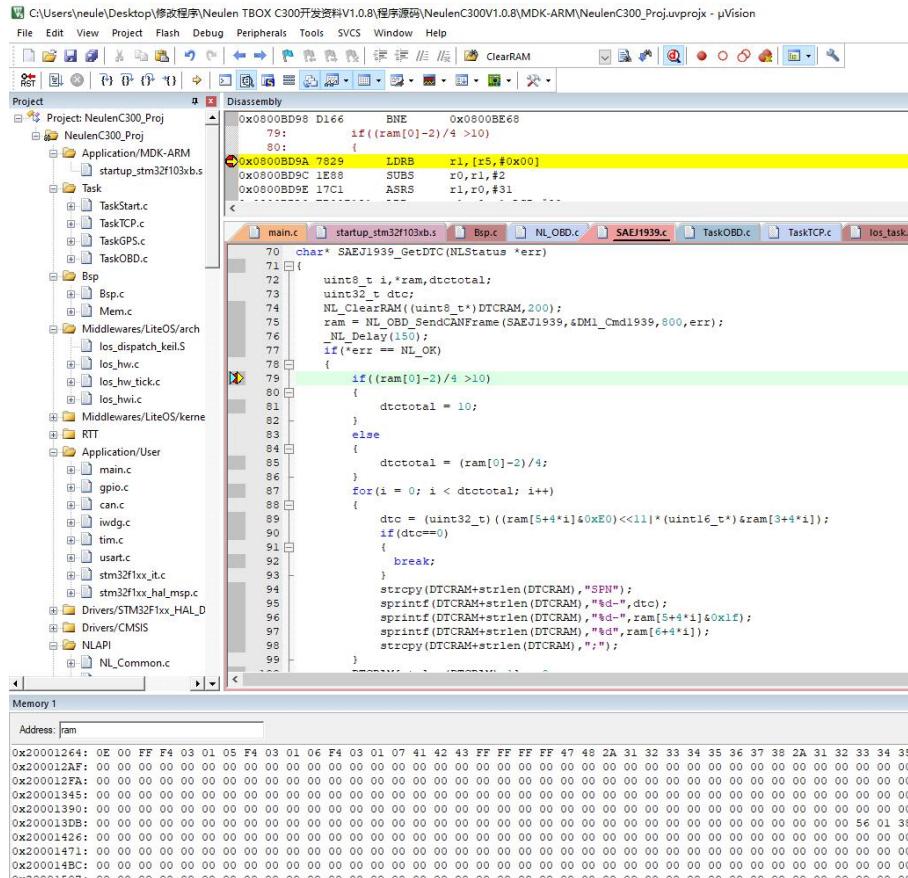
NL_OBD_SendCANFrame 函数处理 BAM 模式的多帧响应。

当我们把“&DM1_Cmd1939”对应的请求数据的目标地址改为全局目标地址 FF 后，如下图所示。

根据上一节内容的讲解，这时候，使用该请求数据获取当前故障码的话，故障码多帧响应将会以 BAM 模式进行响应，下面是 OBD 分析仪采集到的响应数据。



再看下图 NL_OBD_SendCANFrame 函数的返回值。



对比可以看出，调用 NL_OBD_SendCANFrame 函数获取汽车或者模拟器响应数据，无论响应数

据是以 RTS/CTS 模式还是以 BAM 模式进行响应，经过函数 NL_OBD_SendCANFrame 处理后均返回相同的结果，即返回值数据去除了物理层，数据链路层，会话层的内容，直接得到应用层待处理的数据。所以后面我们讲到应用层代码的时候，只要调用了该函数 NL_OBD_SendCANFrame 返回值就可以直接与应用层内容对号入座。

下面来了解下 NL_OBD_SendCANFrame 函数具体实现办法，首先看 NL_OBD_SendCANFrame 原函数代码如下图所示。

```

40 /* **** */
41 * @描述: OBD接口发送一帧CAN数据
42 * @参数: pro: 协议类型, TxMessage: 待发送CAN数据, Timeout: 等待响应时间 , err: 响应是否成功
43 * @返回值: uint8_t* 类型指针, 指向响应数据。
44 */
45 OBDFlagDef OBDFlag;
46 uint8_t CAN_RXRAM[200];
47 uint8_t CAN_PGN[2];
48 uint8_t* NL_OBD_SendCANFrame(ProTypeDef pro, CanTxFrameDef *TxMessage, uint32_t Timeout, NLStatus *err)
49 {
50     uint32_t i;
51     OBDFlag.ProType = pro;
52     if(pro == SAEJ1939)
53     {
54         CAN_PGN[0] = TxMessage->Data[0];
55         CAN_PGN[1] = TxMessage->Data[1];
56     }
57     else if(pro == ISO15765_4STD_500K)
58     {
59         TxMessage->IDE = CAN_ID_STD;
60     }
61     else
62     {
63         TxMessage->IDE = CAN_ID_EXT;
64     }
65     OBDFlag.RxFlag = NL_FAILURE;
66     *err = NL_NOK;
67     _NL_OBD_CAN_Transmit(TxMessage);
68     for (i = 0; i < Timeout; i++)
69     {
70         _NL_Delay(1);
71         if (OBDFlag.RxFlag == NL_SUCCESS)
72         {
73             *err = NL_OK;
74             break;
75         }
76     }
77 }
78 return CAN_RXRAM;
79 }
```

其中 pro 参数是协议类型，当 if 判断 pro 协议类型为 SAEJ1939 时，会把请求数据的数据域的第一和第二字节存储在 CAN_PGN[2] 数组中，因为这两个字节的数据是 PGN。存储完 PGN 之后 OBDFlag.RxFlag 变量设置为 NL_FAILURE。当 OBDFlag.RxFlag 值为 NL_FAILURE 时，表示没有接收到响应数据。当 OBDFlag.RxFlag 值为 NL_SUCCESS 时，表示成功接收到响应数据。所以在发送请求数据之前必须把 OBDFlag.RxFlag 设置为 NL_FAILURE，当接收响应数据成功则直接赋值 NL_SUCCESS 给 OBDFlag.RxFlag 即可，否则保持不变。函数 _NL_OBD_CAN_Transmit 功能是发送一个 CAN 帧，这里发送参数 TxMessage 就是发送请求数据。发送完毕后通过一个 for 循环进行延时等待汽车或者模拟器的响应。for 循环中有一个 _NL_Delay(1) 的 1 毫秒延时函数，也就是 for 循环一次延时 1 毫秒，可以通过参数 Timeout 来设置延时的长短。当 OBDFlag.RxFlag 被赋值为 NL_SUCCESS 时，退出循环，同时给传引用参数*err 赋值 NL_OK，而汽车或者模拟器响应的数据会被存储在数组 CAN_RXRAM[200] 中。如果 for 循环执行完毕，OBDFlag.RxFlag 仍然为 NL_FAILURE，此时*err 值保持不变，值为 NL_NOK，函数执行完毕。所以从前面的代码片段可以看出，在处理该函数返回值的时候，必须要通过判断*err 所指向的值是否为 NL_OK 才能进行处理。

以上函数 NL_OBD_SendCANFrame 源码并不能看出该函数是如何处理响应数据的，函数源码只处理了如何把请求数据发送出去。那么对于响应数据本身是一种需要异步处理的事件，所以我们把这部分处理放在中断中进行。而对于 HAL 库，中断往往被包装成一种事件通过回调函数完成，我们要做的是找到该回调函数，并加入我们需要的处理算法。本函数响应数据是 CAN 总线通信的，所以必须找到 CAN 接收事件的回调函数，该函数就是 HAL_CAN_RxFifo0MsgPendingCallback，代码片段如下图所示。

```

404 L:
405     /* $描述：CAN消息接收回调函数
406     * $参数：CANHandleTypeDef CANhandle
407     * $参数：CANFrameDef RxHeader
408     * $参数：uint8_t RxData[12]
409     */
410     .....
411     if(OBDFlag.ProType == SAEJ1939 && MUXFrameFlagBool == RESET)
412     {
413         uint8_t i;
414         uint8_t CAN_RxRAM[12];
415         CANFrameMaxValue = RxData[0];
416         MUXFrameCounts = 1;
417         TPCN = TPCHAM;
418     }
419     else if(HAL_CAN_0x800Message(CanHandle, CAN_RX_FIFO0, &RxHeader, RxData) == HAL_OK)
420     {
421         if(OBDFlag.ProType == SAEJ1939 && RxData[0] == 0x20 && (RxHeader.ExtId&0xf0000) == 0xEC0000)
422         {
423             CAN_RxRAM[0] = RxData[1];
424             CAN_RxRAM[1] = RxData[3];
425             MUXFrameFlagBool = RESET;
426             MUXFrameCounts = 1;
427             TPCN = TPCHAM;
428         }
429         else if(OBDFlag.ProType == SAEJ1939 && MUXFrameFlagBool == SET && RxData[0] == MUXFrameCount && (RxHeader.ExtId&0xf0000) == 0xE80000 && TPCN == TPCHAM)
430         {
431             for(i = 0; i < 7; i++)
432             {
433                 CAN_RxRAM[i+1] = RxData[i+1];
434             }
435             if(MUXFrameMaxValue == MUXFrameCount)
436             {
437                 MUXFrameFlagBool = RESET;
438                 OBDFlag.RxFlag = NL_SUCCESS;
439             }
440             else
441             {
442                 MUXFrameCounts++;
443             }
444         }
445     }
446     else if(OBDFlag.ProType == SAEJ1939 && RxData[0] == 0x01 && (RxHeader.ExtId&0xf0000) == 0xE00000)
447     {
448         CAN_RxRAM[0] = RxData[1];
449         CAN_RxRAM[1] = RxData[3];
450         MUXFrameMaxValue = RxData[1];
451         MUXFrameFlagBool = SET;
452     }

```

由于篇幅原因并不能拷贝整个回调函数的所有源码展示，如果您是 C300 客户可直接打开下位机工程文件进行阅读，这里我将截图关键部分源码进行详细讲解。

汽车或者模拟器单帧响应函数处理代码如下

```

else if((OBDFlag.ProType == SAEJ1939 && MUXFrameFlagBool == RESET && CAN_PGN[1]<240 && ((RxHeader.ExtId&0xff0000)>>16)==CAN_PGN[1]) ||
       (OBDFlag.ProType == SAEJ1939 && MUXFrameFlagBool == RESET && CAN_PGN[1]>=240 && ((RxHeader.ExtId&0xff0000)>>16)==CAN_PGN[1] && ((RxHeader.ExtId&0xffff00)>>8)==CAN_PGN[0] ))
{
    CAN_RxRAM[0] = RxHeader.DLC;
    for(i = 0; i < RxHeader.DLC; i++)
    {
        CAN_RxRAM[1+i] = RxData[i];
    }
    OBDFlag.RxFlag = NL_SUCCESS;
}

```

该段代码关键在于判断对应请求数据的单帧响应数据。OBDFlag.RxFlag 代表协议类型，此时是 SAEJ1939。MUXFrameFlagBool 代表响应数据是单帧还是多帧，值为 RESET 是单帧，值为 SET 是多帧。CAN_PGN[2]在函数 NL_OBD_SendCANFrame 中已经提前赋值，它存储的是请求数据和响应数据共有的 PGN，主要用于判断响应数据是否是请求数据请求后获得的。因为在实际汽车总线上，前面我们也有提及，往往 OBD 接口是不设置网关的，很多汽车模块的通信数据都集中在 OBD 接口 CAN 总线上通信，所以为了正确过滤掉其它通信模块的数据我们利用 PGN 的判断获取正确的响应数据。判断时，首先要判断是响应数据是 PDU1 格式还是 PDU2 格式，即判断 CAN_PGN[1]是否小于或者大于等于 240，当小于 240 时，响应数据是 PDU1 格式，当大于等于 240 时，响应数据是 PDU2 格式。如果是 PDU1 格式成立，比较 PF 位置数值即可，因为低字节在 CAN 标识符里面被占用为目标地址，所以利用响应数据的 CAN 标识符的 PF 位置比较 CAN_PGN[1]即可，即 $((RxHeader.ExtId&0xff0000)>>16)==CAN_PGN[1]$ 。如果是 PDU2 格式成立不仅要拿响应数据的 CAN 标识符 PF 位置与 CAN_PGN[1]比较，还要拿响应数据的 CAN 标识符 PS 位置与 CAN_PGN[0]比较。 $((RxHeader.ExtId&0xffff00)>>8)==CAN_PGN[0]$ 。通过以上判断确认响应数据是单帧响应数据就可以进行存储在 CAN_RxRAM[200]中了。并且把 OBDFlag.RxFlag 赋值为 NL_SUCCESS。

汽车或者模拟器 BAM 模式响应多帧数据处理代码如下

```

if(OBDFlag.ProType == SAEJ1939 && RxData[0] == 0x20 && (RxHeader.ExtId&0xffff0000) == 0xEC00000)
{
    CAN_RxRAM[0] = RxData[1];
    MUXFrame.MaxValue = RxData[3];
    MUXFrameFlagBool = SET;
    MUXFrameCount = 1;
    TPCM = TPCMBAM;
}
else if(OBDFlag.ProType == SAEJ1939 && MUXFrameFlagBool == SET && RxData[0] == MUXFrameCount && (RxHeader.ExtId&0xffff0000) == 0xEB00000 && TPCM == TPCBM)
{
    for(i = 0; i < 7; i++)
    {
        CAN_RxRAM[i+(RxData[0]-1)*7] = RxData[i+1];
    }
    if(MUXFrame.MaxValue == MUXFrameCount)
    {
        MUXFrameFlagBool = RESET;
        OBDFlag.RxFlag = NL_SUCCESS;
    }
    else
    {
        MUXFrameCount++;
    }
}

```

处理 BAM 模式下的数据响应，可以回顾上节关于协议传输的内容。代码首先要捕抓到 TP.CM_BAM 数据。TP.CM_BAM 数据有两个关键特征。1. PGN 是 60416 也就是 0xEC00。2. 控制字节是 32，也就是 0x20。所以代码有这样的判断 RxData[0]=0x20 && (RxHeader.ExtId&0xffff0000)==0xEC00000。一旦条件成立，程序可以记录更多当前 TP.CM_BAM 的关键信息。它的第二个字节表示有效数据长度(CAN_RxRAM[0]=RxData[1])，第四个字节表示 TP.DT 帧的数量(MUXFrame.MaxValue=RxData[3])。同时设置一些变量为接收 TP.DT 准备，MUXFrameFlagBool 设置为 SET 表示将要接收多帧响应数据。MUXFrameCount=1 用于计数 TP.DT 帧数，同时它与 TP.DT 的序列号对应。TPCM=TPCMBAM 表示当前执行的是 BAM 模式的传输协议。TP.DT 的 PGN 是 0xEB00。所以代码中第二个 if 语句的判断条件就很容易理解，它一旦成立就把 TP.DT 数据上承载的应用层数据存储到 CAN_RxBAM 中。传输完成可以通过 if(MUXFrame.MaxValue==MUXFrameCount) 经行判断。完成后两个变量设置很重要，将 MUXFrameFlagBool 设置为 RESET 表示多帧接收完成，恢复单帧接收。OBDFlag.RxFlag 设置为 NL_SUCCESS 给主程序一个成功响应。

汽车或者模拟器 RTS/CTS 模式响应多帧数据处理代码如下

```

else if(OBDFlag.ProType == SAEJ1939 && RxData[0] == 0x10 && (RxHeader.ExtId&0xffff0000) == 0xEC00000)
{
    CAN_RxRAM[0] = RxData[1];
    MUXFrame.MaxValue = RxData[3];
    MUXFrameFlagBool = SET;
    MUXFrameCount = 1;
    switchvalue = 0;
    switchvalue = RxHeader.ExtId & 0xff;
    switchvalue = (switchvalue<<8) | ((RxHeader.ExtId>>8) & 0xff);
    CTSCmd1939.ExtId = CTSCmd1939.ExtId & (~0xffff);
    CTSCmd1939.ExtId = CTSCmd1939.ExtId | switchvalue;
    CTSCmd1939.Data[1] = MUXFrame.MaxValue;
    CTSCmd1939.Data[5] = RxData[5];
    CTSCmd1939.Data[6] = RxData[6];
    CTSCmd1939.Data[7] = RxData[7];
    EOMACKCmd1939.ExtId = EOMACKCmd1939.ExtId & (~0xffff);
    EOMACKCmd1939.ExtId = EOMACKCmd1939.ExtId | switchvalue;
    for(i=1; i<8; i++)
    {
        EOMACKCmd1939.Data[i] = RxData[i];
    }
    NL_OBD_CAN_Transmit(&CTSCmd1939);
    TPCM = TPCMRTSCTS;
}
else if(OBDFlag.ProType == SAEJ1939 && MUXFrameFlagBool == SET && RxData[0] == MUXFrameCount && (RxHeader.ExtId&0xffff0000) == 0xEB00000 && TPCM == TPCMRTSCTS)
{
    for(i = 0; i < 7; i++)
    {
        CAN_RxRAM[i+(RxData[0]-1)*7] = RxData[i+1];
    }
    if(MUXFrame.MaxValue == MUXFrameCount)
    {
        MUXFrameFlagBool = RESET;
        OBDFlag.RxFlag = NL_SUCCESS;
        _NL_OBD_CAN_Transmit(&EOMACKCmd1939);
    }
    else
    {
        MUXFrameCount++;
    }
}

```

The diagram illustrates the sequence of operations for building frames. A red arrow points from the text block where the TP.CTS frame is being constructed to the text block where the TP.EndOfMsgACK frame is being constructed. This indicates that the data for the TP.CTS frame is used as the basis for the TP.EndOfMsgACK frame.

这部分代码在理解 BAM 模式的代码再看就比较容易理解，它基本思路不变，只是判断值按照协议的定义有所改变，并且与 BAM 广播模式不同，RTS/CTS 是一种握手模式，既然握手，接收到数据之后就必须给发送方答复，所以对比 BAM 模式多出了 TP.CTS 和 TP.EndOfMsgACK 两个答复数据。

第一个 if 判断与 BAM 模式的判断基本一致，只是按照协议定义 TP. RTS 帧的控制字节是 0x10，所以语句 RxData[0]==0x10。一旦条件成立与 BAM 模式一样存储相关特征信息并设置相关变量以便 TP. DT 接收使用。代码片段中我用红色方框和蓝色方框标注了 TP. CTS 帧和 TP. EndOfMsgACK 帧构建的代码，代码只是把 TP. RTS 帧源地址和目标地址调换，数据域部分按照协议定义格式进行填充，填充内容均从 TP. RTS 帧获得。接收 TP. RTS 帧完毕后通过函数 _NL_OBD_CAN_Transmit (&CTSCmd1939) 发送 TP. CTS 帧给汽车或者模拟器确认可进行下一步 TP. DT 数据的通信。接下来第二个 if 判断与 BAM 模式的代码完全一样，但是其中一个判断的变量 TPCM==TPCMRTSCTS 不一样，因为虽然都是接收 TP. DT 数据，数据结构也一致，但是 RTS/CTS 模式下接收完毕要给汽车或者模拟器一个 TP. EndOfMsgACK 帧，以确认都把 TP. DT 数据接收完毕。所以这部分多出发送函数 _NL_OBD_CAN_Transmit (&EOMACKCmd1939)。

在函数 NL_OBD_SendCANFrame 基础上实现 SAEJ1939-71 应用层就相对简单了。下图是我们第一版本软件所实现的三个 OBD 基础功能。



而这三个 OBD 功能对应下位机应用层源码如下图所示。

```

TaskOBD.c ios_task.c TaskOBD.h gpio.h can.h iwdg.c NL_EC20.h includes.h stm32f1xx.h NL_Common.h stm32f1xx_hal_gpio.h

28     if(OBDStruct.SYSValue == SAEJ1939)
29     {
30         if(SAEJ1939_WakeUp() == NL_OK)
31         {
32             NL_LED_ONOFF(LEOBD,ON,Fashing,LED1HZ);
33             while(1)
34             {
35                 if(OBDStruct.VINStruct.flag == RESET)
36                 {
37                     NL_ClearRAM((uint8_t*)OBDStruct.VINStruct.VIN,18);
38                     ram = SAEJ1939_GetVIN(&err);
39                     if(err == NL_OK)
40                     {
41                         strncpy(OBDStruct.VINStruct.VIN,(const char*)ram,17);
42                         OBDStruct.VINStruct.flag = SET;
43                     }
44                 }
45                 if(SAEJ1939_GetNotDrivingState() == NL_OK && OBDStruct.DTCStruct.flag == RESET)
46                 {
47                     ram = SAEJ1939_GetDTC(&err);
48                     if(err == NL_OK)
49                     {
50                         NL_ClearRAM((uint8_t*)OBDStruct.DTCStruct.DTC,100);
51                         strcpy(OBDStruct.DTCStruct.DTC,ram);
52                         OBDStruct.DTCStruct.flag = SET;
53                     }
54                 }
55                 if(OBDStruct.DSStruct.flag == RESET)
56                 {
57                     dsram = SAEJ1939_GetDS(SAEDSItem,&err);
58                     if(err == NL_OK)
59                     {
60                         OBDStruct.LINKSTATUS = SET;
61                         OBDStruct.DSStruct.Total = dsram->Total;
62                         for(i = 0; i < OBDStruct.DSStruct.Total; i++)
63                         {
64                             strncpy(OBDStruct.DSStruct.DS[i],dsram->DS[i],30);
65                         }
66                         OBDStruct.DSStruct.flag = SET;
67                     }
68                 else
69                 {
70                     OBDStruct.LINKSTATUS = RESET;
71                     NL_LED_ONOFF(LEOBD,OFF,Fashing,LED1HZ);
72                     break;
73                 }
74             }
75         }
76     }
77 }

```

下面我们逐个分析这三个 OBD 功能的应用层代码是如何实现的。

1. 读车架号

```

35     if(OBDStruct.VINStruct.flag == RESET)
36     {
37         NL_ClearRAM((uint8_t*)OBDStruct.VINStruct.VIN,18);
38         ram = SAEJ1939_GetVIN(&err);
39         if(err == NL_OK)
40         {
41             strncpy(OBDStruct.VINStruct.VIN,(const char*)ram,17);
42             OBDStruct.VINStruct.flag = SET;
43         }
44     }

```

首先看 TaskOBD 任务里读车架号的这个代码片段，因为 C300 开发板采用了华为轻量级的操作系统，一些变量标志对任务间的调度有至关重要的作用。全局变量标志 OBDStruct.VINStruct.flag 的作用是标识车架号是否读取成功的，当然这个标志是我们自定义的一个变量而非系统变量。在整个代码中当它的值为 RESET 时，表示车架号没有读取成功或者车架号已经上传服务器完毕，而当它的值为 SET 时，表示读取车架号成功，但没有上传服务器。所以对于 TaskTCP 任务来说，只需判断 OBDStruct.VINStruct.flag==SET 便可以上传车架号到服务器，而成功读取的车架号会存储于变量 OBDStruct.VINStruct.VIN 中。上图代码片段很容易理解，35 行判断是否需要读取车架号，37 行代码清空 OBDStruct.VINStruct.VIN 变量，以便存储车架号。38 行获取车架号，当然该函数会用到 NL_OBD_SendCANFrame 函数。返回值便是 17 位车架号存储于 ram 所指向的存储单元。39 行判断读取是否成功，如果成功 41 行把读取到的 17 位车架号存储于 OBDStruct.VINStruct.VIN 变量中，同时在 42 行令 OBDStruct.VINStruct.flag 变量设置为 SET，以便等待 TaskTCP 任务上传至服务器。

下图我们具体看函数 SAEJ1939_GetVIN 源码。

```
45 /* **** * SAEJ1939 获取车架号 **** */
46 * @描述: SAEJ1939获取车架号
47 * @参数: *err :NL_OK:成功 NL_NOK:不成功
48 * @返回值: 车架号存储指针
49 **** */
50 char* SAEJ1939_GetVIN(NLStatus *err)
51 {
52     uint8_t i,*ram;
53     ram = NL_OBD_SendCANFrame(SAEJ1939,&VINCmd1939,800,err);
54     _NL_Delay(150);
55     if(*err == NL_OK)
56     {
57         for(i = 0; i < 17; i++)
58         {
59             VINRAM[i] = ram[i+1];
60         }
61     }
62     VINRAM[17] = 0;
63     return VINRAM;
64 }
```

因为 53 行调用了 NL_OBD_SendCANFrame 函数，应用层就变得尤为简单。NL_OBD_SendCANFrame 函数只要读取成功，通过 err 指针指向的变量值判断是 NL_OK 的话，55 行到 63 行只不过是把返回值指针指向的存储单元存储到 VIMRAM 里而已。前面介绍 NL_OBD_SendCANFrame 函数的时候我们知道，它的返回值指向的存储单元结构由第一个字节的有效字节个数和其后的有效字节构成，所以从代码中可以看出，返回值指向存储单元的第二个字节开始 17 个字节表示车架号。协议我们在前面内容没讲车架号的协议，这里可以补充简单讲解，因为实在简单。参看协议 SAEJ1939-71，车架号 PGN 可以通过上图源码车架号请求数据中找到。如下图所示

在车架号请求数据 VINCmd1939 中，数据域的前两个字节就是它的 PGN，PGN 是 0xFEEC。这时候我们可以直接在 SAEJ1939-71 文档中搜索 FEEC 找到车架号的定义，如下图所示。

PGN 65260 Vehicle Identification		- VI
Byte: 1-n Vehicle Identification Number Delimiter (ASCII "")		
Transmission Repetition Rate:	On request	
Data Length:	Variable	
Extended Data Page:	0	
Data Page:	0	
PDU Format:	254	
PDU Specific:	236	PGN Supporting Information:
Default Priority:	6	
Parameter Group Number:	65260	(0xFFFFE6C)
Start Position	Length	Parameter Name
1	Variable - up to 200 characters ("*" delimited)	Vehicle Identification Number
		SPN
		237

协议中我们看到车架号起始位置就在应用层数据第一个字节开始，也就是说 NL_OBD_SendCANFrame 函数返回值指针指向的存储单元第二字节开始 17 个字节就是车架号，编码格式是以 ASCII 进行编码。

2. 读故障码

首先看下图代码片段

```

45     if(SAEJ1939_GetNotDrivingState() == NL_OK && OBDStruct.DTCStruct.flag == RESET)
46     {
47         ram = SAEJ1939_GetDTC(&err);
48         if(err == NL_OK)
49         {
50             NL_ClearRAM((uint8_t*)OBDStruct.DTCStruct.DTC,100);
51             strcpy(OBDStruct.DTCStruct.DTC,ram);
52             OBDStruct.DTCStruct.flag = SET;
53         }
54     }

```

这部分代码与车架号代码片段类似，这里的 OBDStruct.DTCStruct.flag 用于标记故障码是否读取成功，同时，OBDStruct.DTCStruct.DTC 用于存储故障码。第 45 行代码多出一个判断 SAEJ1939_GetNotDrivingState() == NL_OK，这是用于判断汽车是否处于行驶状态，因为我们的经验中发现一些车型在处于行驶状态下读取故障码的话 ABS 会出现报警灯，所以为了客户更好的使用体验应该在汽车行驶状态下禁止读故障码。47 行调用函数 SAEJ1939_GetDTC 读取故障码，如果读取成功 err 值是 NL_OK。50 行用于清除 OBDStruct.DTCStruct.DTC 变量，51 行把读取到的故障码存储于 OBDStruct.DTCStruct.DTC。最后使 OBDStruct.DTCStruct.flag 为 SET，以便 TaskTCP 知道故障码读取成功可以上传到服务器。

下图我们具体看函数 SAEJ1939_GetDTC 源码。

```

65 /**
66  * @描述: SAEJ1939读取故障码
67  * @参数: *err :NL_OK:成功 NL_NOK:不成功
68  * @返回值: 故障码存储指针
69 */
70 char* SAEJ1939_GetDTC(NLStatus *err)
71 {
72     uint8_t i,*ram,dtctotal;
73     uint32_t dtc;
74     NL_ClearRAM((uint8_t*)DTCRAM,200);
75     ram = NL_OBD_SendCANFrame(SAEJ1939,&DM1_Cmd1939,800,err);
76     _NL_Delay(150);
77     if(*err == NL_OK)
78     {
79         if((ram[0]-2)/4 > 10)
80         {
81             dtctotal = 10;
82         }
83         else
84         {
85             dtctotal = (ram[0]-2)/4;
86         }
87         for(i = 0; i < dtctotal; i++)
88         {
89             dtc = (uint32_t)((ram[5+4*i]&0xE0)<<11|(uint16_t*)&ram[3+4*i]);
90             if(dtc==0)
91             {
92                 break;
93             }
94             strcpy(DTCRAM+strlen(DTCRAM),"SPN");
95             sprintf(DTCRAM+strlen(DTCRAM),"\&d",dtc);
96             sprintf(DTCRAM+strlen(DTCRAM),"\&d",ram[5+4*i]&0x1F);
97             sprintf(DTCRAM+strlen(DTCRAM),"\&d",ram[6+4*i]);
98             strcpy(DTCRAM+strlen(DTCRAM),":");
99         }
100        DTCRAM[strlen(DTCRAM)-1] = 0;
101    }
102    return DTCRAM;
103 }
104

```

75 行通过 NL_OBD_SendCANFrame 函数获取当前故障码应用层数据。如果数据获取成功 79 行到 86 行对故障码个数做限定，如果超过 10 个故障码只显示其中 10 个，如果没超过 10 个故障码就如实显示。故障码数量将被存储在变量 dtctotal 中。值得注意的是计算故障码数量可以通过返回值的第一字节计算得到，即 $((ram[0]-2)/4)$ ，这里可以回顾前面我们讲解 SAEJ1939 协议关于故障码应用层的内容，ram[0] 为返回值有效字节长度，之所以减去 2 是因为故障码应用层数据前 2 个字节分别用于表示故障灯状态以及预留的故障灯状态，并不表示实际故障码，除以 4 是因为 4 个字节表示一个故障码。87 行到 99 行利用 for 循环并且根据 dtctotal 值来逐个把故障码装载到 DTCRAM 中。其中 89 行代码是生成 SPN 值，这里之所以按位与 0xE0 是因为故障码字节第三字节高三位用于表示 SPN 而低 5 位则表示 FMI。这里

对前面协议解析做一个简单回顾。90 到 93 行对生成的 SPN 做一个判断，如果其值为 0 表示故障码有错，直接退出循环。这里用变量名 dtc 其实代表的是 SPN，在当前讨论下 dtc 等于 SPN，FMI 和 OC，命名不算严谨。接下来的 94 行到 98 行利用现成的 C 库函数对故障码做一些格式编辑。100 行代码在故障码装载完毕的 DTCTRAM 内存中的字符最后加入字符结束标志 0。最后获得的故障码存储在 DTCTRAM 数组中，并以指针形式返回。

3. 读数据流

首先看下图代码片段，相对于读车架号和读故障码稍微复杂。

```

55     if(OBDStruct.DSStruct.flag == RESET)
56     {
57         dsram = SAEJ1939_GetDS(SAEDSItem,&err);
58         if(err == NL_OK)
59         {
60             OBDStruct.LINKSTATUS = SET;
61             OBDStruct.DSStruct.Total = dsram->Total;
62             for(i = 0; i < OBDStruct.DSStruct.Total; i++)
63             {
64                 strncpy(OBDStruct.DSStruct.DS[i],dsram->DS[i],30);
65             }
66             OBDStruct.DSStruct.flag = SET;
67         }
68         else
69         {
70             OBDStruct.LINKSTATUS = RESET;
71             NL_LED_ONOFF(LEDOBD,OFF,Fashing,LED1HZ);
72             break;
73         }
74     }

```

55 行判断是否需要读取数据流。57 行获取数据流信息，这里要获取的数据流信息由第一个参数决定，这个参数就是本文档 3.4 章节所讲述的自定义数据流项目结构体，如下所示

```

*****  
* 定义商用车数据流项目  
*****  
DSItemStructDef SAEDSItem = {6,93,83,125,136,33,126};  
*****

```

它的具体运行原理将在下面讲解 SAEJ1939_GetDS 函数原型中具体讲解。57 行函数的返回值就是 SAEDSItem 所标注的索引值对应的数据流读取结果的指针。58 行判断 SAEJ1939_GetDS 函数是否读取成功，这里的判断条件对比读车架号和读故障码有所不同，由 SAEDSItem 决定函数总共发送 6 次请求数据，只要 6 次中有 4 次及以上请求数据失败 err 将等于 NL_NOK，否则是 NL_OK。如果是 NL_NOK，运行 70 到 72 行代码，70 行新增一个全局变量 OBDStruct.LINKSTATUS，它的值是 RESET 时，表示与 OBD 失去连接，它的作用依然是告诉 TaskTCP，此时 TaskTCP 会停止上传 OBD 数据，只上传 GPS 等信息，网站中 OBD 的数据将显示为 NaN，通常用来表示汽车熄火了，71 行是关闭 LED 灯中的 OBD 灯，72 行通过 break 退出当前循环。如果是 NL_OK，运行 60 到 66 行代码，这时候 OBDStruct.LINKSTATUS 给它赋值 SET，让 TaskTCP 正常上传 OBD 数据，并通过 for 循环把函数 SAEJ1939_GetDS 获得的数据流结果赋值到 OBDStruct.DSStruct 中并令 OBDStruct.DSStruct.flag = SET 告知 TaskTCP 可把数据流信息上传服务器。

下图我们具体看 SAEJ1939_GetDS 函数源码。

```

106 *******/
107  * @描述: SAEJ1939读取数据流
108  * @参数:    *err :NOK;成功 NL_NOK;不成功
109  * @返回值: 数据流存储结构
110 */
111 DSStructRef* SAEJ1939_GetDS(DSItemStructDef item,NLStatus *err)
112 {
113     uint8_t i,j,*ram,*p;
114     uint32_t data32;
115     DSStruct.Total = item.Total;
116     for(i = 0; i < item.Total; i++)
117     {
118         ReqCmd1939.Data[0] = (uint8_t)(DSControl1939[item.Item[i]].PGN & 0xFF);
119         ReqCmd1939.Data[1] = (uint8_t)(DSControl1939[item.Item[i]].PGN>>8 & 0xFF);
120         ram = NL_OBD_SendCANFrame(SAEJ1939,&ReqCmd1939,800,err);
121         _NL_Delay(150);
122         if(*err == NL_OK)
123         {
124             ErrorCount = 0;
125             if(DSControl1939[item.Item[i]].Type == Numeric)
126             {
127                 data32 = 0;
128                 p = (u8*)&data32;
129                 for(j = 0; j < DSControl1939[item.Item[i]].DataLength; j++)
130                 {
131                     p[j] = ram[DSControl1939[item.Item[i]].X+i+j];
132                 }
133                 sprintf(DSStruct.DS[i],DSControl1939[item.Item[i]].Format,DSControl1939[item.Item[i]].Equation0(data32));
134             }
135             else
136             {
137                 strcpy(DSStruct.DS[i],"");
138                 DSControl1939[item.Item[i]].Equation1(DSStruct.DS[i]+strlen(DSStruct.DS[i]),&ram[DSControl1939[item.Item[i]].X+i],SET,item.Item[i]);
139                 strcpy(DSStruct.DS[i]+strlen(DSStruct.DS[i]),"\0");
140             }
141         }
142         else
143         {
144             if (++ErrorCount > 4)
145             {
146                 *err = NL_NOK;
147                 return &DSStruct;
148             }
149         }
150     }
151     return &DSStruct;
152 }

```

函数中 item 作为 SAEDSItem 的形式参数，114 行将 SAEDSItem 第一个成员 Total, 也就是数据流总数赋值给结构体变量 DSStruct 中的成员 Total, 所以我们知道，整个函数的工作就是把读到的数据流结果赋值给 DSStruct，所以 115 行通过 for 循环赋值给 DSStruct，每循环一次赋值一个数据流结果给 DSStruct，循环次数由 SAEDSItem.Total 的值决定。117 行和 118 行是给请求数据赋值 PGN，关于请求数据的帧结构在前面协议讲解中已经具体分析过。这里可以找到程序中 ReqCmd1939 定义内容如下图所示。

```

/*****************************************命令*****************************************/
CanTxFrameDef DM1_Cmd1939 = {0, 0x18EAFFFF, CAN_ID_EXT, CAN_RTR_DATA, 3, 0xCA, 0xFE, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
CanTxFrameDef VINCmd1939 = {0, 0x18EA00F9, CAN_ID_EXT, CAN_RTR_DATA, 3, 0xEC, 0xFE, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
CanTxFrameDef ReqCmd1939 = {0, 0x18EA00F9, CAN_ID_EXT, CAN_RTR_DATA, 3, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
CanTxFrameDef CTSCmd1939 = {0, 0x1CEC00F9, CAN_ID_EXT, CAN_RTR_DATA, 8, 0x11, 0x00, 0x01, 0xFF, 0xFF, 0x00, 0x00, 0x00};
CanTxFrameDef EOMACKCmd1939 = {0, 0x1CEC00F9, CAN_ID_EXT, CAN_RTR_DATA, 8, 0x13, 0x00, 0x01, 0xFF, 0xFF, 0x00, 0x00, 0x00};
/*****************************************数据*****************************************/

```

深色部分数据就是数据流请求数据，但是这个请求数据我们发现它的 PGN 部分的值是 0。函数 117 行和 118 行就是给这个数据的数据域第一字节和第二字节赋值 PGN 的。119 行利用 NL_OBD_SendCANFrame 函数完成当前 PGN 下的数据流请求和响应数据的采集。121 行通过对 *err 判断知道 NL_OBD_SendCANFrame 函数是否获得汽车的响应。如果没有获得响应就有可能本车型不支持该数据流那么就执行 143 行到 147 行代码，这个代码很容易看明白 ErrorCount 是一个错误计数器，如果 NL_OBD_SendCANFrame 函数错误一次就计数 1，如果连续错误到达 4 次后就被认为该车已经熄火或者已经断开连接，这时候就给本函数的 err 参数所指向的存储单元赋值 NL_NOK。那么 121 行如果判断是成功的就执行 123 行到 139 行代码，123 行代码就是给 ErrorCount 计数器清 0 的，如果连续错误没有到达 4 次那么被认为这个车是可以正常通信的只不过其中有个别数据流不支持罢了。124 行判断当前数据流是数值型数据流还是文本型数据流，这个概念在我们的 HD-OBD 模拟器中提到过，是我们定义的而非协议提及。当前 C300 开发板第一版本软件上位机只能显示数值型的数据流，而文本型数据流可以通过 TCP 数据传送到服务器，但是服务器程序不做处理显示，所以如果您是自行开发服务器程序的就可以利用上文本型数据流。数值型数据流程序中用字符 Numeric 表示，文本型数据流程序中用字符 Character 表示。函数解释到这里必须理解结构体变量 DSControl1939，下图是 DSControl1939 变量定义并赋值的代码片段。

```

/*
*文件: SAEJ1939_71.c
*作者: Tony.Neulen
*日期: 2019/10/17 10:07
*描述: SAEJ1939_71应用层数据流实现
*产品购买:https://shop115932063.taobao.com
*****/

#include "includes.h"

//DSTotal1939

const DSControl1939TypeDef DSControl1939[DSTotal1939] = {
    {Numeric ,0xFFE0,0,4,NONE,"%1f" , "km" , function01, NONE}, //000.行程"
    {Numeric ,0xFFE0,4,4,NONE,"%1f" , "km" , function01, NONE}, //001.总里程"
    {Numeric ,0xFE8E,0,2,NONE,"%1f" , "kW" , function02, NONE}, //002.额定的发动机能量"
    {Numeric ,0xFEE8,2,2,NONE,"%2f" , "rpm" , function01, NONE}, //003.额定的发动机速度"
    {Numeric ,0xFEC7,0,1,NONE,"%0f" , "%" , function03, NONE}, //004.转换杠换档位置"
    {Numeric ,0xFEC7,1,1,NONE,"%0f" , "%" , function03, NONE}, //005.转换杠轨道位置"
    {Character,0xFEC7,2,1, 0,"" , "" , NONE,CHfunction01}, //006.空挡指示器"
    {Character,0xFEC7,2,1, 2,"" , "" , NONE,CHfunction01}, //007.衔接指示器"
    {Character,0xFEC7,2,1, 4,"" , "" , NONE,CHfunction01}, //008.中心轨道指示器"
    {Character,0xFEC7,3,1, 0,"" , "" , NONE,CHfunction01}, //009.轨道激励器1"
    {Character,0xFEC7,3,1, 2,"" , "" , NONE,CHfunction01}, //010.换档激励器1"
    {Character,0xFEC7,3,1, 4,"" , "" , NONE,CHfunction01}, //011.轨道激励器2"
    {Character,0xFEC7,3,1, 6,"" , "" , NONE,CHfunction01}, //012.换档激励器2"
    {Character,0xFEC7,4,1, 0,"" , "" , NONE,CHfunction01}, //013.范围内高激励器"
    {Character,0xFEC7,4,1, 2,"" , "" , NONE,CHfunction01}, //014.范围内低激励器"
    {Character,0xFEC7,4,1, 4,"" , "" , NONE,CHfunction01}, //015.努力直接激励器"
    {Character,0xFEC7,4,1, 6,"" , "" , NONE,CHfunction01}, //016.努力非直接激励器"
    {Character,0xFEC7,5,1, 0,"" , "" , NONE,CHfunction01}, //017.离合器激励器"
    {Character,0xFEC7,5,1, 2,"" , "" , NONE,CHfunction01}, //018.锁定离合器激励器"
    {Character,0xFEC7,5,1, 4,"" , "" , NONE,CHfunction01}, //019.减燃料激励器"
    {Character,0xFEC7,5,1, 6,"" , "" , NONE,CHfunction01}, //020.惯性刹车激励器"
    {Numeric ,0xFE81,0,2,NONE,"%1f" , "m^3/h" , function04, NONE}, //021.发动机燃油流量1"
    {Numeric ,0xFE81,2,2,NONE,"%1f" , "m^3/h" , function04, NONE}, //022.发动机燃油流量2"
    {Numeric ,0xFE81,4,1,NONE,"%0f" , "%" , function03, NONE}, //023.发动机燃油阀1位置"
    {Numeric ,0xFE81,5,1,NONE,"%0f" , "%" , function03, NONE}, //024.发动机燃油阀2位置"
    {Numeric ,0xFE81,6,1,NONE,"%0f" , "%" , function03, NONE}, //025.发动机请求燃油阀1位置"
    {Numeric ,0xFE81,7,1,NONE,"%0f" , "%" , function03, NONE}, //026.发动机请求燃油阀2位置"
    {Numeric ,0xFE89,0,2,NONE,"%4f" , "%" , function05, NONE}, //027.理想的额定的排氧量"
    {Numeric ,0xFE89,2,2,NONE,"%4f" , "%" , function05, NONE}, //028.理想的排氧量"
    {Numeric ,0xFE89,4,2,NONE,"%4f" , "%" , function05, NONE}, //029.实际排氧量"
    {Numeric ,0xFE89,6,1,NONE,"%0f" , "%" , function06, NONE}, //030.发动机排气氧传感器加油修正"
}

```

为了能够理解这个结构体变量赋值内容的意义，可以找到 DSControl1939TypeDef 类型声明的结构体的定义，如下图所示。

```

typedef struct
{
    __IO DataType Type; //数据类型
    u16 PGN;           //PGN
    u8 X;              //参数字节开始的位置 (Start Position)
    u8 DataLength;     //参数长度 (Data Length)
    u8 Bit;            //参数位
    char *Format;      //输出格式控制
    char* unit;         //单位
    double (*Equation0)(int data); //数值型计算公式
    ERRORTYPE (*Equation1)(char* data1,u8* data2,FlagStatus flag,u8 num); //文本型计算公式
} DSControl1939TypeDef;
 *****/

```

为了能很好理解每个成员的意义，下面分别举两个例子。

```

{Numeric ,0xF004,2,1,NONE,"%0f" , "%" , function06, NONE}, //092.实际发动机-扭矩百分比"
{Numeric ,0xF004,3,2,NONE,"%3f" , "rpm" , function01, NONE}, //093.发动机转速"
{Numeric ,0xF004,5,1,NONE,"%0f" , "%" , function07, NONE}, //094.针对发动机控制的控制设备源地址"
{Character,0xF004,6,1, 0,"" , "" , NONE,CHfunction04}, //095.发动机启动器模式"
{Numeric ,0xF004,7,1,NONE,"%0f" , "%" , function06, NONE}, //096.发动机命令-扭矩百分比"

```

1. 数值型数据流举例，发动机转速。

数据类型: Numeric //表示发动机转速是数值型数据

PGN: 0xF004 //表示发动机转速的 PGN

参数字节开始的位置: 3 //表示发动机转速的有效字节是在响应数据的第三字节开始。

参数长度: 2 //表示发动机转速有效字节有 2 个字节表示。

参数位: NONE //表示此处不需要填写。

输出格式控制: “%.3f” //表示发动机转速的数值是浮点型, 精确到小数点后三位。

单位: “rpm” //表示发动机转速单位。

数值型计算公式: function01 //表示发动机转速计算函数的地址, 它的作用就是把汽车响应的数据计算成发动机转速显示值。下面是它的函数源码。

```
double function01(int x)
{
    return x*0.125;
}
```

它的意义就是取 PGN 为 0xF004 的汽车响应数据的第三字节开始的两个字节乘以 0.125 获得发动机转速的显示值。这里值得注意的是程序的计数通常是从 0 开始计数的, 所以这里取的第三字节其实是第四字节, 这与文本表达不一样, 所以值得注意。

文本型计算公式: NONE //数值型数据流不会用到此值, 填 NONE 即可。

以下我们截图 SAEJ1939-71 对照下刚才填写的内容是否与协议一致。

PGN 61444 <i>Electronic Engine Controller 1</i>			- EEC1
Engine related parameters			
Transmission Repetition Rate:			engine speed dependent
Data Length:	8		
Extended Data Page:	0		
Data Page:	0		
PDU Format:	240		
PDU Specific:	4	PGN Supporting Information:	
Default Priority:	3		
Parameter Group Number:	61444	(0xF004)	
Start Position	Length	Parameter Name	SPN
1.1	4 bits	Engine Torque Mode	899
2	1 byte	Driver's Demand Engine - Percent Torque	512
3	1 byte	Actual Engine - Percent Torque	513
4-5	2 bytes	Engine Speed	190
6	1 byte	Source Address of Controlling Device for Engine Control	1483
7.1	4 bits	Engine Starter Mode	1675
8	1 byte	Engine Demand - Percent Torque	2432

这里的 Start Position 就是 【参数字节开始的位置: 3】 , 为什么不写 4, 刚才已经说了程序保持了从 0 开始计数, 而协议文档是从 1 开始计数, 所以协议填入程序需要减 1. Length 2byte 就是【 参数长度: 2 】。下面在看 function01 函数的由来。

SPN 190 *Engine Speed*

Actual engine speed which is calculated over a minimum crankshaft angle of 720 degrees divided by the number of cylinders.

Data Length: 2 bytes
Resolution: 0.125 rpm/bit, 0 offset

Data Range: 0 to 8,031.875 rpm Operational Range: same as data range

Type: Measured

Supporting information:

PGN reference: 61444

函数中的 return x*0.125; 算法由来就是协议定义的 Resolution 0.125 rpm/bit, 0 offset。

2. 文本型数据流举例, 发动机启动器模式。

数据类型: Character //表示发动机启动器模式数据流是文本类型的数据。

PGN: 0xF004 //表示发动机启动器模式数据流的 PGN, 它与发动机转速是共用 PGN 的。

参数字节开始的位置: 6 //表示发动机启动器模式有效字节从响应数据的第六字节开始取。

参数长度: 1 //表示发动机启动模式有效字节是 1 个字节。

参数位:0 //表示发动机启动模式有效位是第 0 位开始有效。

输出格式控制：“ ” //因为是文本型数据不需要格式控制，用空格文本表示。

单位：“ ” //因为是文本型数据不需要单位，用空格文本表示。

数值型计算公式：NONE

文本型计算公式：CHfunction04 //表示发动机启动模式计算公式，这个公式把原始有效数据转换成显示的文本，下面我们看下它的函数源码。

```
ERRORType CHfunction04(char* data1, u8* data2, FlagStatus flag, u8 num)
{
    if(flag == RESET)
    {
        *data2 |= crol(0x00,DSControl1939[num].Bit);
        if (!strcmp((const char *)data1,"start not requested"))
        {
            *data2 |= crol(0x00,DSControl1939[num].Bit);
        }
        else if (!strcmp((const char *)data1,"starter active, gear not engaged"))
        {
            *data2 |= crol(0x01,DSControl1939[num].Bit);
        }
        else if (!strcmp((const char *)data1,"starter active, gear engaged"))
        {
            *data2 |= crol(0x02,DSControl1939[num].Bit);
        }
        else if (!strcmp((const char *)data1,"start finished; starter not active after having been actively engaged"))
        {
            *data2 |= crol(0x03,DSControl1939[num].Bit);
        }
        else if (!strcmp((const char *)data1,"starter inhibited due to engine already running"))
        {
            *data2 |= crol(0x04,DSControl1939[num].Bit);
        }
        else if (!strcmp((const char *)data1,"starter inhibited due to engine not ready for start (preheating)"))
        {
            *data2 |= crol(0x05,DSControl1939[num].Bit);
        }
        else if (!strcmp((const char *)data1,"starter inhibited due to driveline engaged or other transmission inhibit"))
        {
            *data2 |= crol(0x06,DSControl1939[num].Bit);
        }
        else if (!strcmp((const char *)data1,"starter inhibited due to active immobilizer"))
        {
            *data2 |= crol(0x07,DSControl1939[num].Bit);
        }
    }
}
```

图 A

```
if ((*data2&crol(0x0F,DSControl1939[num].Bit)) == crol(0x00,DSControl1939[num].Bit))
{
    strcpy(data1,"start not requested");
}
else if ((*data2&crol(0x0F,DSControl1939[num].Bit)) == crol(0x01,DSControl1939[num].Bit))
{
    strcpy(data1,"starter active, gear not engaged");
}
else if ((*data2&crol(0x0F,DSControl1939[num].Bit)) == crol(0x02,DSControl1939[num].Bit))
{
    strcpy(data1,"starter active, gear engaged");
}
else if ((*data2&crol(0x0F,DSControl1939[num].Bit)) == crol(0x03,DSControl1939[num].Bit))
{
    strcpy(data1,"start finished; starter not active after having been actively engaged (after 50ms mode goes to 0000)");
}
else if ((*data2&crol(0x0F,DSControl1939[num].Bit)) == crol(0x04,DSControl1939[num].Bit))
{
    strcpy(data1,"starter inhibited due to engine already running");
}
else if ((*data2&crol(0x0F,DSControl1939[num].Bit)) == crol(0x05,DSControl1939[num].Bit))
{
    strcpy(data1,"starter inhibited due to engine not ready for start (preheating)");
}
else if ((*data2&crol(0x0F,DSControl1939[num].Bit)) == crol(0x06,DSControl1939[num].Bit))
{
    strcpy(data1,"starter inhibited due to driveline engaged or other transmission inhibit");
}
else if ((*data2&crol(0x0F,DSControl1939[num].Bit)) == crol(0x07,DSControl1939[num].Bit))
{
    strcpy(data1,"starter inhibited due to active immobilizer");
}
else if ((*data2&crol(0x0F,DSControl1939[num].Bit)) == crol(0x08,DSControl1939[num].Bit))
{
    strcpy(data1,"starter inhibited due to starter over-temp");
}
else if ((*data2&crol(0x0F,DSControl1939[num].Bit)) == crol(0x09,DSControl1939[num].Bit))
{
    strcpy(data1,"Reserved");
}
```

图 B

因为源码过长这里截图分为图 A 和图 B, 图 A 源码其实在本程序中没有起作用，因为这个函数在 HD-OBD 模拟器和 C300 开发板中同时使用的，图 A 源码只在 HD-OBD 模拟器起作用，所以图 A 源码我们可以不看，看图 B。图 B 源码只看红色方框 if 判断，看懂就完全理解了，首先参数 data2 代表的是汽车响应的有效数据，这里就是 PGN 为 0xF004 的汽车响应数据的第 6 字节。Data1 是处理后要显示的文本信息。crol(0x01,DSControl1939[num].Bit)意思是将 0x01 做移 DSControl1939[num].Bit 位，这时候 DSControl1939[num].Bit 的值就是刚才我们填的 0，也就左移了 0 位。(*data2&crol(0x0F,DSControl1939[num].Bit))就是拿采集到是数据用它的低四位比较，低 4 位是 0x01 的话那么条件满足 data1 返回文本内容"starter active, gear not engaged"。

以下我们截图 SAEJ1939-71 对照下刚才填写的内容是否与协议一致。

PGN 61444		Electronic Engine Controller 1	- EEC1
Engine related parameters			
Transmission Repetition Rate:	engine speed dependent		
Data Length:	8		
Extended Data Page:	0		
Data Page:	0		
PDU Format:	240		
PDU Specific:	4	PGN Supporting Information:	
Default Priority:	3		
Parameter Group Number:	61444 (0xF004)		
Start Position	Length	Parameter Name	SPN
1.1	4 bits	Engine Torque Mode	899
2	1 byte	Driver's Demand Engine - Percent Torque	512
3	1 byte	Actual Engine - Percent Torque	513
4-5	2 bytes	Engine Speed	190
6	1 byte	Source Address of Controlling Device for Engine Control	1483
7.1	4 bits	Engine Starter Mode	1675
8	1 byte	Engine Demand – Percent Torque	2432

【参数字节开始的位置: 6】一样的处理，程序从 0 开始计数所以协议是 7 的填入程序是 6.

【参数位:0】也和字节计数一样，文档是 7.1 也就是第 7 字节第 1 位，而程序计数从 0 开始就改成第 6 字节第 0 位，所以参数位这个地方填写 0. 再看下计算函数由来。

SPN 1675 Engine Starter Mode	
There are several phases in a starting action and different reasons why a start cannot take place.	
0000 start not requested	
0001 starter active, gear not engaged	
0010 starter active, gear engaged	
0011 start finished; starter not active after having been actively engaged (after 50ms mode goes to 0000)	
0100 starter inhibited due to engine already running	
0101 starter inhibited due to engine not ready for start (preheating)	
0110 starter inhibited due to driveline engaged or other transmission inhibit	
0111 starter inhibited due to active immobilizer	
1000 starter inhibited due to starter over-temp	
1001-1011 Reserved	
1100 starter inhibited - reason unknown	
1101 error (legacy implementation only, use 1110)	
1110 error	
1111 not available	
Data Length:	4 bits
Resolution:	16 states/4 bit, 0 offset
Data Range:	0 to 15
Type:	Status
Supporting information:	
PGN reference:	61444
Operational Range:	same as data range

理解了 DSControl1939 结构体变量后，我们再看函数 SAEJ1939_GetDS 的 122 行到 140 行如何处理数据流的代码就变得简单了。首先 124 行通过索引值和 DSControl1939 结构体的关系判断当前读的数据流是数值型数据流还是文本型数据流，如果是数值型数据流执行 125 行到 133 行代码，如果是文本型数据流执行 135 行到 139 行代码。其中数值型数据流处理办法看 126 行到 131 行将响应数据的取值加载到 data32 中计算，data32 是 uint32_t 类型，总共可以加载 4 个字节的数据，所以这里可以看出做诊断程序用 32 位单片机的重要性。最后在 132 行将 data32 传入计算函数并进行格式控制存储在 DSStruct 结构体中。136 行到 138 行文本数据流的处理过程基本类似，只是最后计算函数传导出来的结果是文本数据。

7.1.3 SAEJ1939 协议以及在开发板中的程序实现视频教程

SAEJ1939 协议解读视频教程

视频教程链接: <https://www.bilibili.com/video/BV1iE411H7dK?p=9>

Neulen 工作室 <https://shop115932063.taobao.com>

SAEJ1939 协议代码实现视频教程（上集）

视频教程链接: <https://www.bilibili.com/video/BV1iE411H7dK?p=10>

SAEJ1939 协议代码实现视频教程（下集）

视频教程链接: <https://www.bilibili.com/video/BV1iE411H7dK?p=11>

7.2 ISO15765-4 协议以及在开发板中的程序实现

7.2.1 ISO15765-4 协议解读

前面我们已经解读过 SAEJ1939 协议，我们接下来依然按照前面的协议解读办法，围绕 协议通信如何唤醒、数据格式、通信信息管理和应用层如何定义这四方面内容进行解读协议。首先对 ISO15765-4 协议做一个基本了解，如下图所示。

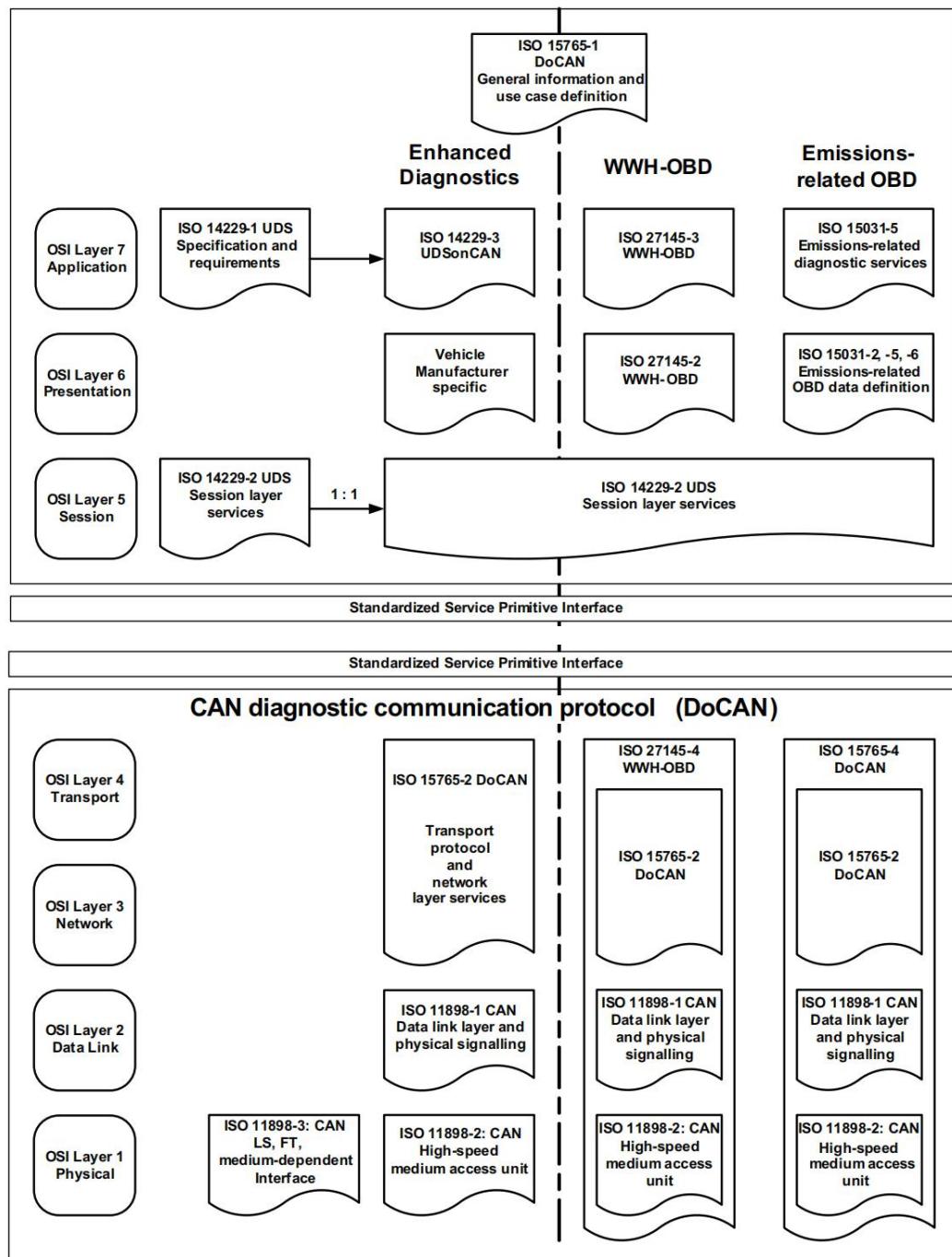


Figure 1 — Diagnostic communication over CAN document reference according to OSI model

此图源自 ISO15765-4 协议，它告诉我们学习 CAN 诊断协议可以根据这个 OSI 模型图提供的文献进行学习。这里它提到了三种基于 CAN 的诊断协议，Enhanced Diagnostics(统一诊断服务), WWH-OBD (全球统一的重型发动机的车载诊断系统)、Emissions-related OBD(排放协议)。我们的 C300 开发板使用的协议正是第三种排放协议，也就是所谓的 OBD2 协议。在前面我已经讲解了我们做这种协议的必要性。除此之外两种协议将来我们再具体分析，这里做个大概了解，Enhanced Diagnostics 本人也是接触最多的协议之一，目前大部分的汽车厂自定义协议都使用这种协议形式，注意的是这种协议不具备直接拿来开发产品的条件，因为上图中可以看到从描述层开始标注了“Vehicle manufacturer specific(汽车制造商特定的)”，也就是说，它从描述层开始是由各个汽车厂根据自己的车型自定义的协议，必须和汽车厂合作获得描述层和应用层协议才具备开发产品的条件。还有 WWH-OBD 这是一个比较新的协议规则，目前我还没有接触到这种协议，或是接触过了误把它认为是 Enhanced Diagnostics 协议。从图中可以看出，这三种协议在 OSI 模型中从物理层到会话层都是一样的，实际操作会发现它们除了应用层具体值定义不一样，其它都完全一致，所以我们这里学习了排放协议后基本就掌握了另外两种协议规则。

1. 协议通信如何唤醒？

在解读 SAEJ1939 协议的时候提到过，对于基于 CAN 物理总线的协议是不需要唤醒的，但是多个 CAN 物理总线的协议如何区分当前汽车用的是哪个 CAN 物理总线协议类型就成了个问题。ISO15765-4 协议里给了我们一个办法，如下面截图所示。

6 External test equipment initialization sequence

6.1 General

The external test equipment shall support the initialization sequence specified in this part of ISO 15765. See Figure 2.

The purpose of the external test equipment initialization sequence is to automatically detect whether the vehicle supports legislated OBD or WWH-OBD on CAN using the physical layer specified in Clause 12.

Furthermore, the initialization sequence determines the communication compliance status of vehicles by analysing their responses to:

- ISO 15031-5 service 0x01 0x00 (PID supported) requests, or
- ISO 27145-3 service 0x22 0xF810 (DID protocol identification) request with a positive response.

Only vehicles that follow the WWH-OBD regimen will have ECUs that reply to the functional request service 0x22 DID 0xF810 for protocol identification. Vehicles that respond only to the functional request service 0x01 PID 0x00 support earlier OBD communication methods. Vehicles that do not respond to either request do not support regulated OBD diagnostics under this part of ISO 15765. Subclause 6.3 describes this procedure.

For each legislated OBD/WWH-OBD service that requires the determination of “supported” information, the external test equipment has to update its list of expected responding legislated OBD/WWH-OBD ECUs prior to any data parameter requests. For applicable services see either ISO 15031-5 (for legislated OBD) or ISO 27145-3 (for legislated WWH-OBD).

The external test equipment initialization sequence supports single baudrate initialization (e.g. 500 kBit/s) and multiple baudrate initialization (e.g. 250 kBit/s and 500 kBit/s) and is separated into the following tests:

- a) 11 bit CAN identifier validation, and
- b) 29 bit CAN identifier validation.

在它的第六章专门描述了外部测试设备如何初始化序列的问题。文字很多但是总结就两点内容，我分别用红色方框圈出来了。第一点内容是 使用 ISO15031-5 服务字节为 01 PID 为 00，进行请求获得一个汽车积极响应。第二点内容是 如果是单波特率的使用 500K 速率进行初始化，而多波特率的话使用 250K 和 500K 进行初始化，并且初始化的同时采用 11 位标识符和

29 位标识符进行验证。而实际应用中我们发现，ISO15765-4 OBD2 排放协议实车并不存在 250K 速率的。所以 C300 开发板只采用了 500K 速率，主要目的为了减少自动搜索协议的时间。

下面我们看下 C300 开发板关于初始化命令的截图。

```

5 /* ************************************************************************
6 * @文件: ISO15765_4.c
7 * @作者: Tony.Neulen
8 * @日期: 2019/10/17 10:07
9 * @描述: ISO15765_4 下的函数实现
10 * @产品购买:https://shop115932063.taobao.com
11 *****/
12 #include "includes.h"
13
14
15 /* **** 系统激活 ****/
16 CanTxFrameDef EntCmd15765 = {0x7DF, 0x18DB33F1, CAN_ID_STD, CAN_RTR_DATA, 8, 0x02, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};01 00
17 /* **** 读取车架号 ****/
18 CanTxFrameDef DTCCmd15765 = {0x7DF, 0x18DB33F1, CAN_ID_STD, CAN_RTR_DATA, 8, 0x01, 0x03, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
19 /* **** 读取数据流 ****/
20 CanTxFrameDef VinCmd15765 = {0x7DF, 0x18DB33F1, CAN_ID_STD, CAN_RTR_DATA, 8, 0x02, 0x09, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00};
21 /* **** 读取服务字节 ****/
22 CanTxFrameDef DSCmd15765 = {0x7DF, 0x18DB33F1, CAN_ID_STD, CAN_RTR_DATA, 8, 0x02, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
23 */

```

如图所示，红色框就是初始化命令，按照协议的要求 C300 开发板以 500K 速率发送这个初始化命令以等待汽车积极响应。当然命令要分别以 11 位标识符和 29 位标识符进行请求，并且命令内容要搭载服务字节 0x01 和 PID 00。其实 00 被叫做 PID supported 表示数据流支持的情况，关于这个内容我们会在应用层协议具体讲解。这里的 11 位标识符是 0x7df, 29 位标识符是 0x18DB33F1。那么这里问题来了，我怎么知道是这两个标识符的值呢？引入下面第二个问题。数据格式？

2. 数据格式？

一帧 CAN 数据在物理层以上往往讨论的是标识符和数据域两部分，协议中往往称一帧 CAN 数据为 PDU，这在前面的 SAEJ1939 协议中被大家所知。这里我们要解决 PDU 构成问题，才能理解后面的协议内容。

首先看 CAN 标识符的定义，学习 CAN 总线的时候我们都知道，标识符其主要作用还是标识网络节点，即这个数据源自哪个节点。所以它的定义可以从 ISO15765-4 协议的网络层中找到相关定义。在 ISO15765-4 协议中，标识符在协议功能上被分为两类，一类是功能性的（Functional），另一类是物理性的（Physical）。它们的功能如下图所示。

Table 5 — Definition of diagnostic addresses versus type of CAN identifier

CAN identifier	Target address (TA)	Source address (SA)	TA type (TAtype)	Message type (Mtype)
Functional request	Legislated OBD/WWH-OBD system = 0x33	External test equipment = 0xF1	functional	diagnostics
Physical response	External test equipment = 0xF1	Legislated OBD/WWH-OBD ECU = 0XX	physical	diagnostics
Physical request	Legislated OBD/WWH-OBD ECU = 0XX	External test equipment = 0xF1	physical	diagnostics
0XX ECU physical diagnostic address				
NOTE	For detailed descriptions of parameters TA, SA, TAtype and Mtype, see ISO 15765-2.			

如果标识符的目标地址为 0X33，源地址是 0xF1，这时候的搭载该标识符的数据被称为功能性的请求，当然这个数据是什么我们先不管，这涉及到数据域搭载内容的问题。

如果目标地址是 0xF1, 源地址是 ECU 物理诊断地址 0xXX, 搭载该标识符的数据被称为物理响应。

如果目标地址是 ECU 物理诊断地址 0xXX, 源地址是 0xF1, 搭载标识符的数据被称为物理请求。

ISO15765-4 协议的标识符有 11 位和 29 位, 对于 11 位标识符这里存在一个问题, 11 位只有一个字节多三位的容量是无法同时装载源地址和目标地址的。这就是为什么在定义 CAN1.0 标准基础上加入了 29 位扩展标识符作为 CAN2.0 标准。同时, ISO15765-4 对 11 位标识符的功能性请求标识符, 物理性请求标识符和物理性响应标识符做了具体定义, 如下图所示。

10.5.2 11 bit CAN identifiers

Table 6 specifies the 11 bit CAN identifiers for legislated OBD/WWH-OBD, based on the defined mapping of the diagnostic addresses.

Table 6 — 11 bit legislated OBD/WWH-OBD CAN identifiers

CAN identifier	Description
0x7DF	CAN identifier for functionally addressed request messages sent by external test equipment
0x7E0	Physical request CAN identifier from external test equipment to ECU #1
0x7E8	Physical response CAN identifier from ECU #1 to external test equipment
0x7E1	Physical request CAN identifier from external test equipment to ECU #2
0x7E9	Physical response CAN identifier from ECU #2 to external test equipment
0x7E2	Physical request CAN identifier from external test equipment to ECU #3
0x7EA	Physical response CAN identifier from ECU #3 to external test equipment
0x7E3	Physical request CAN identifier from external test equipment to ECU #4
0x7EB	Physical response CAN identifier ECU #4 to the external test equipment
0x7E4	Physical request CAN identifier from external test equipment to ECU #5
0x7EC	Physical response CAN identifier from ECU #5 to external test equipment
0x7E5	Physical request CAN identifier from external test equipment to ECU #6
0x7ED	Physical response CAN identifier from ECU #6 to external test equipment
0x7E6	Physical request CAN identifier from external test equipment to ECU #7
0x7EE	Physical response CAN identifier from ECU #7 to external test equipment
0x7E7	Physical request CAN identifier from external test equipment to ECU #8
0x7EF	Physical response CAN identifier from ECU #8 to external test equipment
While not required for current implementations, it is strongly recommended (and may be required by applicable legislation) that for future implementations, the following 11 bit CAN identifier assignments be used:	
— 0x7E0/0x7E8 for ECM (engine control module);	
— 0x7E1/0x7E9 for TCM (transmission control module).	

图中第一行 0x7DF CAN 标识符通过外部测试设备发送功能性寻址请求信息。目标地址是 0x33 , 源地址是 0xF1 的功能性请求标识符。这里就解决了前面我们的疑问, 初始化命令的 11 位标识符是怎么来的问题。因为 C300 开发板做的是法规排放协议 OBD2, 它的目标地址就一定是 0x33。

接下来若干行罗列了发动机控制模块 (ECU) 从 1 到 8 需要使用的物理性请求标识符和物理性响应标识符的定义。我们在实际测试中发现一般一辆车会存在 1 个到 2 个 ECU。如果是两个 ECU 的汽车, 当我们发送请求数据的时候会得到 0x7e8 和 0x7e9 两个物理性响应数据。这时候我们应该取数值较小的 0x7e8 作为取值而放弃 0x7e9, 这是参考一些大型 OBD 设备厂商

的做法，比如博世公司的诊断仪。有一点需要注意，以上两个表格有功能性请求却没有功能性响应。因为协议要求，功能性请求后应该获得的是物理性响应数据。

接下来再看 29 位扩展 CAN 标识符的定义，如下图所示。

Table 7 — Summary of 29 bit CAN identifier format — Normal fixed addressing

CAN ID bit position	28	24	23	16	15	8	7	0
Functional CAN identifier	0x18		0xDB		TA		SA	
Physical CAN identifier	0x18		0xDA		TA		SA	
NOTE The CAN identifier values given in this table use the default value for the priority information in accordance with ISO 15765-2.								

Table 8 — 29 bit legislated OBD/WWH-OBD CAN identifiers

CAN identifier	Description
0x18 DB 33 F1	Functional request CAN identifier from external test equipment to ECUs with #33
0x18 DA XX F1	Physical request CAN identifier from external test equipment to ECU #XX
0x18 DA F1 XX	Physical response CAN identifier from ECU #XX to external test equipment
While not required for current implementations, it is strongly recommended (and may be required by applicable legislation) that for future implementations, the physical ECU addresses be used in accordance with the assignments found in SAE J2178/1.	

Table7 描述了 29 位 CAN 标识符的基本结构，具体结构在 ISO15765-2 协议中有具体定义，对于应用来说，此表已经足够满足要求，0x18DB(TASA) 用于标识功能性 CAN 标识符，0x18DA(TASA) 用于标识物理性 CAN 标识符。而具体用于请求还是响应只需修改目标地址(TA)和源地址(SA)即可。如 Table8 所示，标识符 0x18DB33F1 用于功能性请求，标识符 0x18DAXXF1 用于物理性请求，标识符 0x18DAF1XX 用于物理性响应。到此我们完全找到了关于初始化命令中分别使用的 11 位 CAN 标识符 0x7DF 和 29 位 CAN 标识符 0x18DB33F1。

对于 ECU 物理地址 0XX, Table8 最后一句话告诉了我们它的分配表格可以参阅 SAEJ2178-1 协议如下图所示。

TABLE 11—NODE PHYSICAL ADDRESS ASSIGNMENTS

Node Category	Address (hex)	# of Nodes
Powertrain Controllers:	00 - 1F	32
Integration/Manufacturer Expansion	00 - 0F	16
Engine Controllers	10 - 17	8
Transmission Controllers	18 - 1F	8
Chassis Controllers:	20 - 3F	32
Integration/Manufacturer Expansion	20 - 27	8
Brake Controllers	28 - 2F	8
Steering Controllers	30 - 37	8
Suspension Controllers	38 - 3F	8
Body Controllers:	40 - C7	136
Integration/Manufacturer Expansion	40 - 57	24
Restraints	58 - 5F	8
Driver Information/Displays	60 - 6F	16
Lighting	70 - 7F	16
Entertainment/Audio	80 - 8F	16
Personal Communications	90 - 97	8
Climate Control (HVAC)	98 - 9F	8
Convenience (Doors, Seats, Windows, etc.)	A0 - BF	32
Security	C0 - C7	8
Electric Vehicle Energy Transfer System (EV-ETS)	C8 - CB	4
Utility Connection Services (FN#1)	C8	1
AC to AC Conversion (FN#2)	C9	1
AC to DC Conversion (FN#3)	CA	1
Energy Storage Management (FN#4)	CB	1
Future Expansion:	CC - CF	4
Manufacturer Specific:	D0 - EF	32
Off-Board Testers/Diagnostic Tools:	F0 - FD	14
All Nodes:	FE	1
Null Node:	FF	1

而实际开发中我多以实车测试为准，该表格它只定义了一个区间范围，具体使用哪个由制造商决定。实际测试中我们以功能性标识符请求汽车，汽车多以 0x10 的源地址进行响应，所以我们的模拟器均默认发动机源地址是 0x10。

接下来再看数据域的定义，在 ISO15765-2 协议对数据域格式有详细定义。补充一点就是通常我们要了解一个协议的数据格式往往是看它的数据链路层，对于本协议来说，数据链路层被定义在 ISO11898-1 协议，同时部分定义于网络层协议 ISO15765-2。由于现在我们使用的芯片基本都集成了 CAN 控制器，学习使用芯片的时候其实我们已经了解了 ISO11898-1 协议的内容，所以不推荐大家浪费时间去看这个协议，直接看网络层 ISO15765-2 协议关于数据域格式的定义即可。其中该协议对数据格式定义如下图所示。

Table 5 — N_PDU format

Address information	Protocol control information	Data field
N_AI	N_PCI	N_Data

其中，N_AI 地址信息就是刚才我们讨论的标识符内容。数据域的信息就由两部分组成 N_PCI 和 N_Data 组成。N_PCI 称为协议控制信息，这和传输协议息息相关。N_Data 就是应用层数据，应用层数据被定义在 ISO15031 协议，它可以承载在这里的 ISO15765 协议，也可以承载在 ISO14230, ISO9141 等协议上。

这里关键内容是 N_PCI，N_PCI 使用不同的值，CAN 数据帧（PDU）作用就不一样。如下图所示

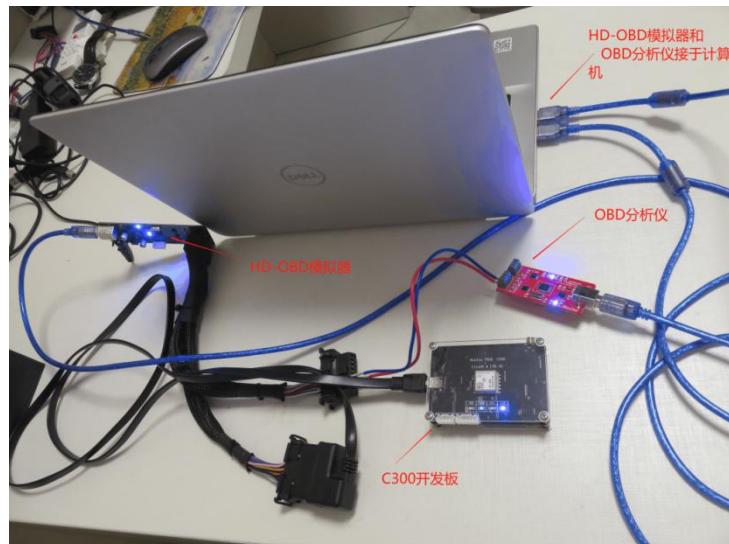
Table 3 — Summary of N_PCI bytes

N_PDU name	N_PCI bytes			
	Byte #1		Byte #2	Byte #3
	Bits 7 – 4	Bits 3 – 0		
SingleFrame (SF)	N_PCltype = 0	SF_DL	N/A	N/A
FirstFrame (FF)	N_PCltype = 1		FF_DL	N/A
ConsecutiveFrame (CF)	N_PCltype = 2	SN	N/A	N/A
FlowControl (FC)	N_PCltype = 3	FS	BS	STmin

标识 N_PCI 功能作用主要是 CAN 帧（PDU）数据域的前面 3 个字节，而区分 N_PCI 类型的只有 CAN 帧（PDU）数据域的第 1 字节的高 4 位。如图所示，其值为 0, 1, 2, 3 分别代表单独帧 SingleFrame, 首帧 FirstFrame, 邻帧 ConsecutiveFrame, 流控帧 FlowControl。为了让您理解这些不同类型 CAN 帧（PDU）的功能和作用，我们不得不把本协议数据格式的数据域格式放放，先了解下本协议的下一个问题，本协议的协议通信信息管理。

3. 通信信息管理

这时候我们直接使用 C300 开发板连接 OBD 模拟器，并用 OBD 分析仪采集数据进行举例讲解。连接方法如下图所示。



这里提示一点就是，如果您购买我们C300 开发板套餐里涵盖 OBD 模拟器，OBD 模拟器默认程序是商用车 SAEJ1939 模拟程序，使用 ISO15765-4 等乘用车模拟程序需要进行固件切换至汽油车模拟软件和使用对应上位机软件操作。

下面我们进行一些操作获取 ISO15765-4 原始数据。

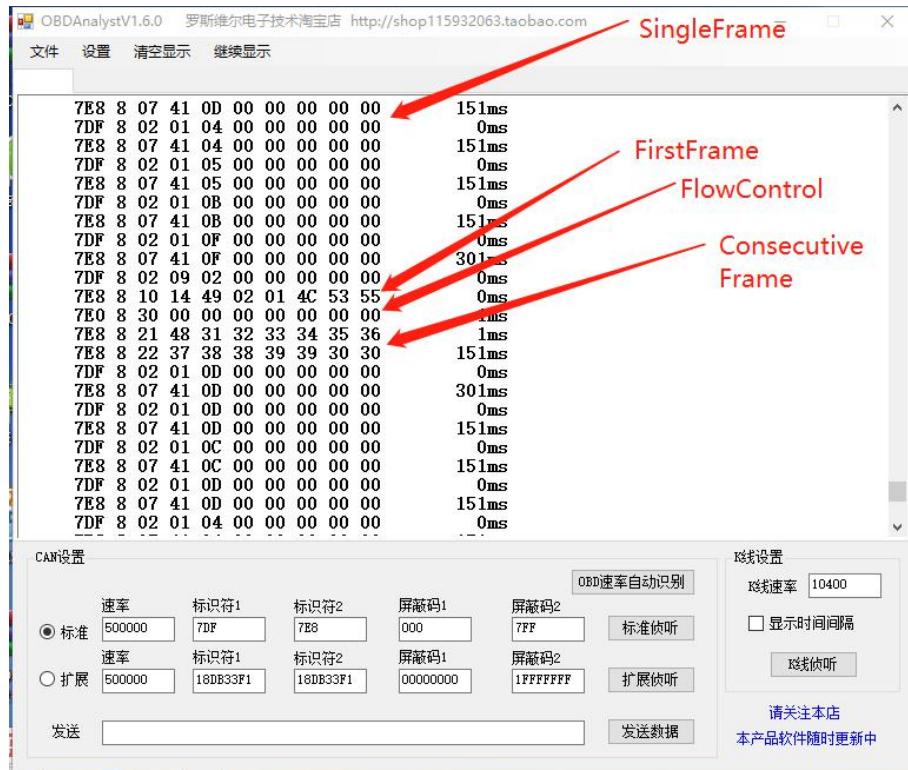
首先，打开 OBD 模拟器上位机软件，选择 ISO15765-4(11BIT ID 500K) 协议，这是标准 CAN 协议。



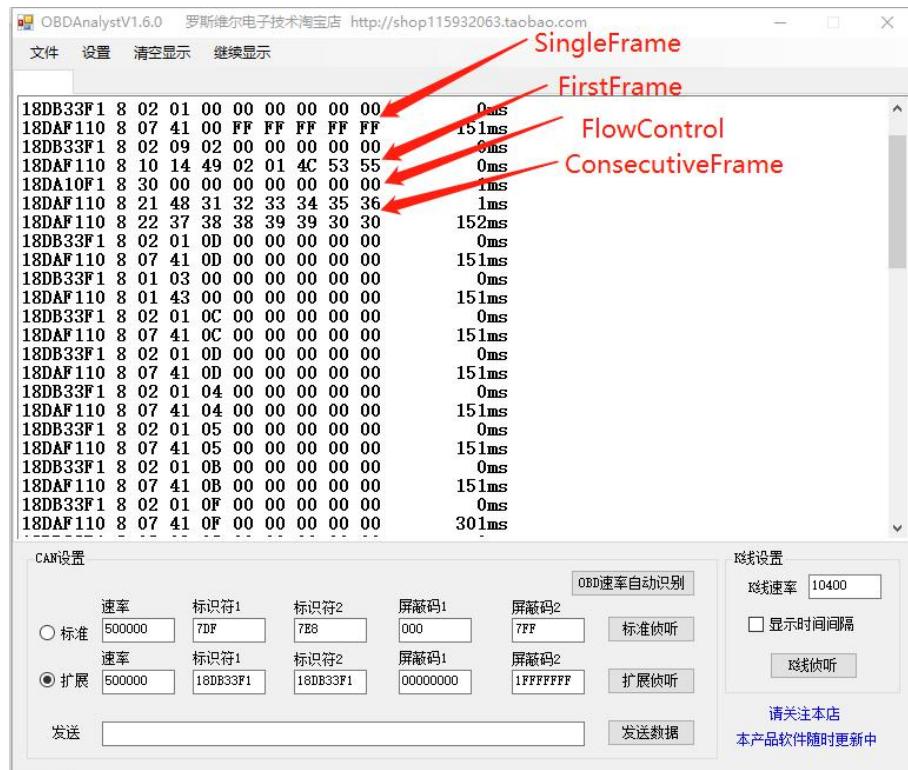
然后，任意设置一个车架号，主要是为了讲解数据超过 8 个字节多帧传输使用。



最后，C300 开发板通信过程中，使用 OBD 分析仪对数据进行采集如下所示。



为了对比方便，我设置模拟器协议为 ISO15765-4 (29BIT 500K)，并使用 OBD 分析仪采集数据如下所示。



一帧 CAN 数据就能承载的信息使用 SingleFrame，需要超过一个 CAN 帧承载的信息就必须使用其它的传输方式进行分割。接下来很快就要讲到。下面是它的传输模型。

[Figure 3](#) shows an example of an unsegmented message transmission.

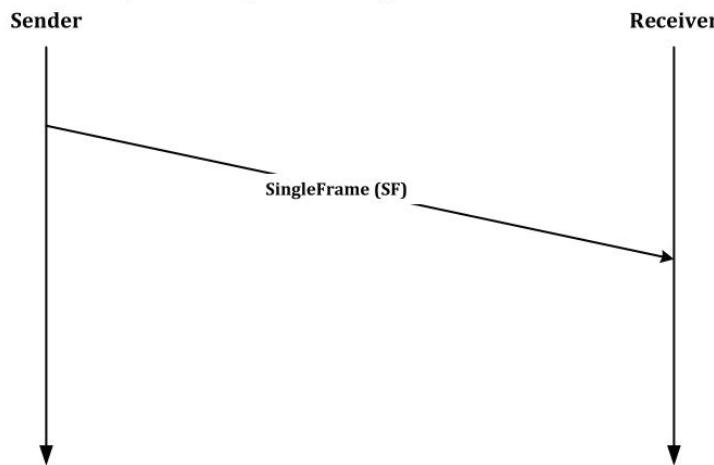


Figure 3 — Example of an unsegmented message

发送者向接收者发送一帧 CAN 数据，就这么简单，其实关键点在于它的数据域格式，下图红色方框我们看 SingleFrame 数据域格式。

Table 3 — Summary of N_PCI bytes

N_PDU name	N_PCI bytes		
	Byte #1 Bits 7 – 4	Byte #2 Bits 3 – 0	Byte #3
SingleFrame (SF)	N_PCItype = 0	SF_DL	N/A
FirstFrame (FF)	N_PCItype = 1	FF_DL	N/A
ConsecutiveFrame (CF)	N_PCItype = 2	SN	N/A
FlowControl (FC)	N_PCItype = 3	FS	BS
			STmin

第一字节高四位 N_PCItype 是 0 的话就表示 SingleFrame，而第一字节低四位 SF_DL 全称 Single frame data length in bytes（单帧数据字节长度）。平时客户询问的时候我通常称它为有效字节长度。下面我们直接看上面 OBD 分析仪采集到的数据进行举例。

7DF 8 02 01 04 00 00 00 00 00
7E8 8 07 41 04 00 00 00 00 00

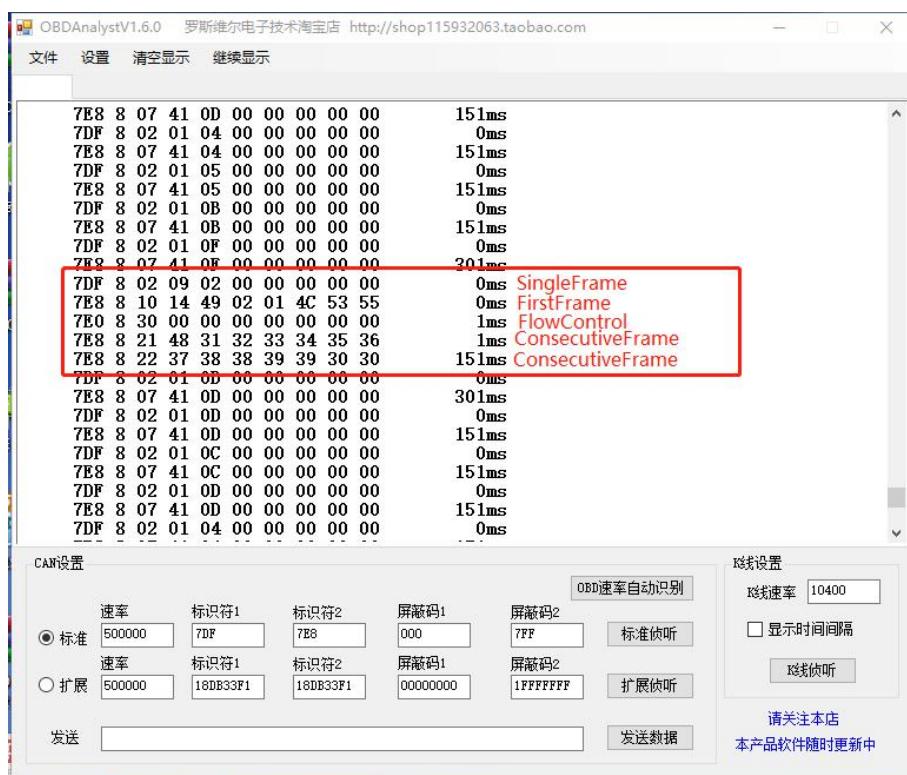
这是标准 CAN，也就是 ISO15765-4(11BIT 500K) 协议里通信数据，关于标识符的讲解我们在数据格式里具体讲过了。这里简单提一下，首先 7DF 是功能性标识符，表示其目标地址是 0x33，显然这是 C300 开发板作为外部诊断设备向汽车 ECU 的功能性请求。而汽车 ECU 也做了响应，响应物理性地址 7E8。这里无论是 C300 开发板向汽车请求的数据还是汽车响应给 C300 开发板的数据都采用了 SingleFrame，因为它们数据域的第一字节高四位均为 0。它们低四位的值表示有效字节长度，分别是 02 和 07，也就是说标识符 7DF 请求命令承载的信息是 01 04，同理标识符 7E8 的响应数据承载的信息就是从 41 开始的 7 个字节。至于这些承载的信息具体表示什么，我们会在解读应用层协议再具体讲解，这里我们知道这些承载的信息就是应用层的原始数据。

同理，如果是扩展 CAN，也就是 ISO15765-4(29BIT 500K) 协议，如下图所示。

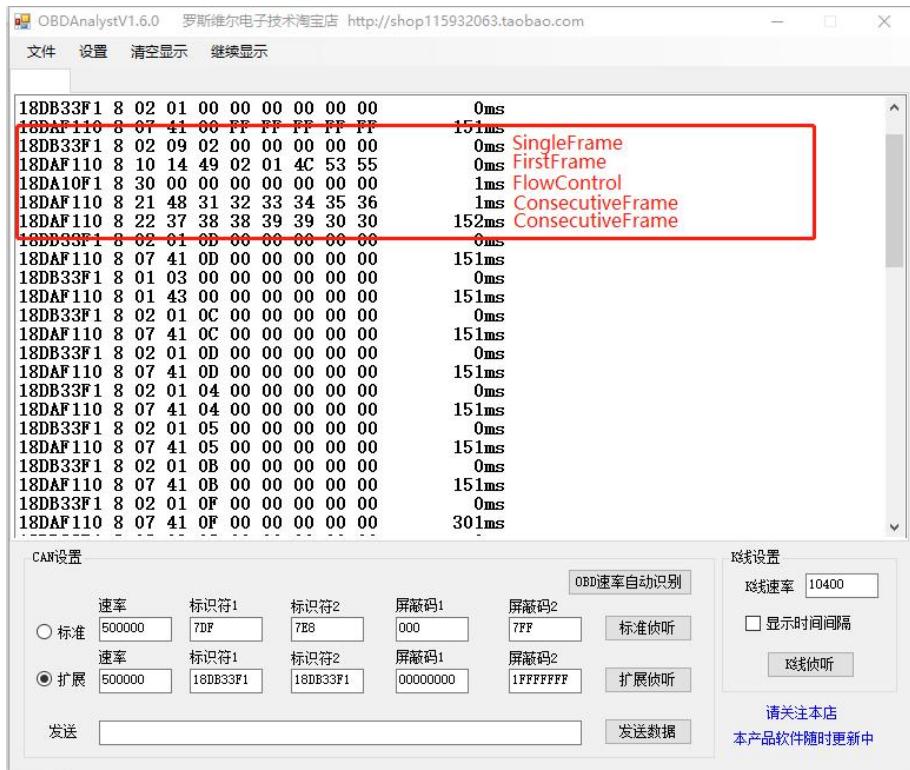
18DB33F1	8	02	01	00	00	00	00	00	00
18DAF110	8	07	41	00	FF	FF	FF	FF	FF

只是标识符不同，数据域规则是完全一致的。这里补充一点就是如上图 C300 开发板以功能性请求发送数据，其数据目标地址是 0x33，但是响应数据则以物理性响应，其源地址是 0x10。也就是说，虽然 C300 开发板以 0x33 目标地址请求了，但是得到的响应数据其源地址不是 0x33，而是实际物理地址 0x10（发动机控制单元）进行响应，响应数据不存在 0x33 的源地址。

如果应用层需要承载的信息超过 7 个字节，就必须把数据进行分割而采用多帧传输的通信协议进行传输。为什么说是 7 个字节呢？我们看 SingleFrame 第一个字节被用来表示网络协议控制信息类型 (N_PCItype) 和有效字节长度，剩下所能承载的信息长度就只有 7 个字节了。显然在汽车诊断里面很多情况，信息长度是大于 7 个字节的，比如读故障码，比如读车架号等，上面 OBD 分析仪采集 C300 开发板通信数据中，我们故意设置一个车架号让 C300 开发板读取车架号实际举例说明如何进行分割传输数据。如下图所示。



ISO15765-4(11BIT 500K) 协议数据



ISO15765-4 (29BIT 500K) 协议数据

这里读车架号的请求数据仍然是以 SingFrame 发送请求。因为接下来讲解应用层协议的时候我们就会知道，读车架号的请求数据只有两个字节有效信息，09 表示服务字节（SID），02 是信息类型（INFOTYPE）。

而车架号的响应数据就没那么简单了，因为要承载 17 位车架号对应的 ASCII 编码总共需要 17 个字节数据，所以必须进行分割多帧传输。用到的网络协议控制信息类型（N_PCItype）包括了 FirstFrame, FlowControl, ConsecutiveFrame。帧格式和传输协议通信过程可以对照下面两个图进行。

Table 3 — Summary of N_PCI bytes

N_PDU name	N_PCI bytes			
	Byte #1		Byte #2	Byte #3
Bits 7 – 4	Bits 3 – 0			
SingleFrame (SF)	N_PCItype = 0	SF_DL	N/A	N/A
FirstFrame (FF)	N_PCItype = 1	FF_DL	N/A	
ConsecutiveFrame (CF)	N_PCItype = 2	SN	N/A	N/A
FlowControl (FC)	N_PCItype = 3	FS	BS	STmin

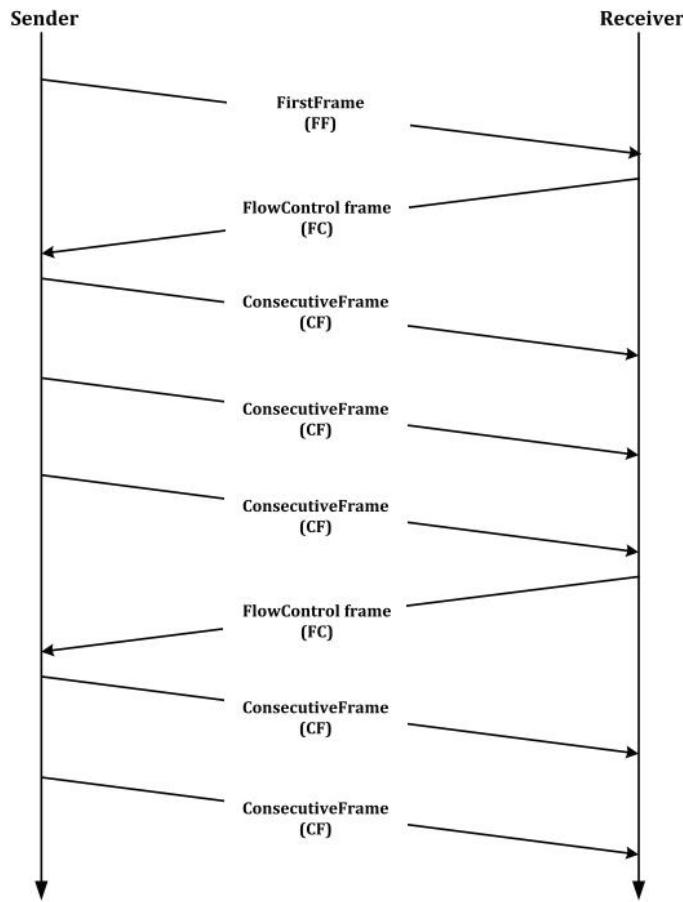


Figure 4 — Example of a segmented message

汽车响应 C300 开发板读取车架号请求过程是这样的，如 Figure4 图中所示，首先响应 FirstFrame，这个帧的格式就是数据域第一字节高四位是 1。接下来的低四位和第二字节总共 12 位 (Bit) 表示有效字节长度 (FF_DL)。FirstFrame 除掉这两个信息字节只能承载 6 个字节车架号 ASCII 编码。C300 开发板在接收到 FirstFrame 后，必须发送 FlowControlFrame 以指定接下来的数据如何传输。FlowControlFrame 的标识符值得注意不能以功能性标识符进行发送，而是从 FirstFrame 数据中得知目标地址，以物理性标识符进行发送。如我们采集到的数据，以标准 CAN 通信的数据为例，C300 开发板得知 FirstFrame 的标识符为 7E8，那么就知道响应 ECU 是 ECU#1，外部诊断设备发送的请求物理性标识符就必须是 7E0，所以可以看到 C300 开发板此时发送的 FlowControlFrame 为 7E0 8 30 00 00 00 00 00 00 00。同理 扩展 CAN 通信的数据，FirstFrame 源地址是 0x10，那么 C300 开发板则以 0x10 为目标地址发送物理性标识符请求，发送的 FlowControlFrame 为 18DA10F1 8 30 00 00 00 00 00 00 00。FlowControlFrame 数据域第一字节高四位为 3 表示 FlowControl，而低四位表示 FS (FlowStats)，值为 0 表示继续发送，其它值比如 1 是等待，通常我们做程序就是通过流控帧继续发送，所以第一字节用最多的就是 0x30。低四位其它值代表意义如下表所示。

Table 18 — Definition of FS values

Value	Description
0_{16}	ContinueToSend (CTS) The FlowControl ContinueToSend parameter shall be encoded by setting the lower nibble of the N_PCI byte #1 to "0". It shall cause the sender to resume the sending of ConsecutiveFrames. The meaning of this value is that the receiver is ready to receive a maximum of BS number of ConsecutiveFrames.
1_{16}	Wait (WAIT) The FlowControl Wait parameter shall be encoded by setting the lower nibble of the N_PCI byte #1 to "1". It shall cause the sender to continue to wait for a new FlowControl N_PDU and to restart its N_BS timer. If FlowStatus is set to Wait, the values of BS (BlockSize) and ST_min (SeparationTime minimum) in the FlowControl message are not relevant and shall be ignored.
2_{16}	Overflow (OVFLW) The FlowControl Overflow parameter shall be encoded by setting the lower nibble of the N_PCI byte #1 to "2". It shall cause the sender to abort the transmission of a segmented message and make an N_USData.confirm service call with the parameter <N_Result> = N_BUFFER_OVFLW. This N_PCI FlowStatus parameter value is only allowed to be transmitted in the FlowControl N_PDU that follows the FirstFrame N_PDU and shall only be used if the message length FF_DL of the received FirstFrame N_PDU exceeds the buffer size of the receiving entity. If FlowStatus is set to Overflow, the values of BS (BlockSize) and ST_min (SeparationTime minimum) in the FlowControl message are not relevant and shall be ignored.
$3_{16} - F_{16}$	Reserved This range of values is reserved by this part of ISO 15765.

FlowControlFrame 数据域第二字节表示 BS (BlockSize)，其实就是告诉发送端接收到流控帧后发送端应该发送多少帧 ConsecutiveFrame，如果 BS 是 0，表示一次性发送完所有 ConsecutiveFrame，比如我们当前车架号剩下 11 字节车架号没有传送，还需要两帧 ConsecutiveFrame 承载。BS=0 时，告诉汽车 ECU 一次性发完两帧 ConsecutiveFrame。如果 BS=1，则汽车 ECU 接收到流控帧后，只发送一帧 ConsecutiveFrame。剩下的一帧必须等待 FlowControlFrame 的再次请求才能发送。BS 具体描述如下图表格所示。

Table 19 — Definition of BS values

Value	Description
00_{16}	BlockSize (BS) The BS parameter value 0 shall be used to indicate to the sender that no more FC frames shall be sent during the transmission of the segmented message. The sending network layer entity shall send all remaining ConsecutiveFrames without any stop for further FC frames from the receiving network layer entity.
$01_{16} - FF_{16}$	BlockSize (BS) This range of BS parameter values shall be used to indicate to the sender the maximum number of ConsecutiveFrames that can be received without an intermediate FC frame from the receiving network entity.

FlowControlFrame 数据域第三字节表示 STmin 表示 ConsecutiveFrame 之间间隔时间，单位是 MS。通常性能较好的芯片这个值设置为 0 即可，如果您使用的芯片性能较低，可以酌情设置该值，取值范围如下图表格所示。

Table 20 — Definition of STmin values

Value	Description
$00_{16} - 7F_{16}$	SeparationTime minimum (ST_min) range: 0 ms - 127 ms The units of ST_min in the range $00_{16} - 7F_{16}$ (0 - 127) are absolute milliseconds (ms).
$80_{16} - F0_{16}$	Reserved This range of values is reserved by this part of ISO 15765.
$F1_{16} - F9_{16}$	SeparationTime minimum (ST_min) range: 100 μ s - 900 μ s The units of ST_min in the range $F1_{16} - F9_{16}$ are even multiples of 100 μ s, where parameter value $F1_{16}$ represents 100 μ s and parameter value $F9_{16}$ represents 900 μ s.
$FA_{16} - FF_{16}$	Reserved This range of values is reserved by this part of ISO 15765.

所以 C300 开发板无论是标准 CAN 还是扩展 CAN 使用的流控帧分别是 7E0 8 30 00 00 00 00 00 00 和 18DA10F1 8 30 00 00 00 00 00 00 00，当然标识符会根据实际情况，程序自

动调整。流控帧都要求汽车 ECU 一次性把所有数据传输到 C300 开发板，并且 ConsecutiveFrame 之间的时间间隔没有要求。

汽车接收到流控帧信息之后，接下来就是发送两帧 ConsecutiveFrame 承载最后 11 个字节的车架号。ConsecutiveFrame 第一字节高四位值为 2，低四位的则是一个序列号，从 0x0 开始计数到 0xf，然后再从 0 开始计数，以此循环。第一帧 ConsecutiveFrame 第一字节是 0x21，第二帧 ConsecutiveFrame 第一字节是 0x22，一直发送 ConsecutiveFrame 的话，会计数到 0x2f，然后再从 0x20 计数。ConsecutiveFrame 除了第一字节外，剩下 7 个字节都是承载应用层信息。

Table 17 — Definition of SN values

Value	Description
0 ₁₆ – F ₁₆	SequenceNumber (SN) The SequenceNumber (SN) shall be encoded in the lower nibble bits of N_PCI byte #1. The SN shall be set to a value within the range of 0 to 15.

至此，应用层以下的协议解读已经足够我们编写诊断程序非应用层内容了，下面我们来解读应用层协议。

4. 应用层如何定义

接下来所解读的应用层协议主要是 ISO15031 这几个协议，ISO15031 有 7 个协议，如下图所示。

- Part 1: General information
- Part 2: Terms, definitions, abbreviations and acronyms
- Part 3: Diagnostic connector and related electrical circuits, specification and use
- Part 4: External test equipment
- Part 5: Emissions-related diagnostic services
- Part 6: Diagnostic trouble code definitions
- Part 7: Data link security

它们分别定义什么内容很清晰，按照我们的学习风格只看对我们项目有用 的协议，其它可以暂时不用看。其中红色框第五部分和第六部分是我们必须要研究的。第五部分是 ISO15031-5 定义排放相关的诊断服务，第六部分是 ISO15031-6 是定义诊断故障码的。功能性描述的协议主要还是看 ISO15031-5 协议，如下图所示 ISO15031-5 目录。

5	Technical requirements	4
5.1	General requirements.....	4
5.2	Diagnostic service requirements	4
5.3	Diagnostic message format.....	25
5.4	Allowance for expansion and enhanced diagnostic services	29
5.5	Definition of PIDs for Services \$01 and \$02.....	29
5.6	Format of data to be displayed.....	29
6	Diagnostic service definition for ISO 9141-2, ISO 14230-4, and SAE J1850	31
6.1	Service \$01 — Request current powertrain diagnostic data.....	31
6.2	Service \$02 — Request powertrain freeze frame data.....	35
6.3	Service \$03 — Request emission-related diagnostic trouble codes.....	39
6.4	Service \$04 — Clear/reset emission-related diagnostic information	44
6.5	Service \$05 — Request oxygen sensor monitoring test results	46
6.6	Service \$06 — Request on-board monitoring test results for specific monitored systems.....	51
6.7	Service \$07 - Request emission-related diagnostic trouble codes detected during current or last completed driving cycle	56
6.8	Service \$08 — Request control of on-board system, test or component.....	57
6.9	Service \$09 — Request vehicle information	60
7	Diagnostic service definition for ISO 15765-4	73
7.1	Service \$01 — Request current powertrain diagnostic data.....	73
7.2	Service \$02 — Request powertrain freeze frame data.....	79
7.3	Service \$03 — Request emission-related diagnostic trouble codes.....	84
7.4	Service \$04 — Clear/reset emission-related diagnostic information	87
7.5	Service \$05 — Request oxygen sensor monitoring test results	88
7.6	Service \$06 — Request on-board monitoring test results for specific monitored systems.....	89
7.7	Service \$07 — Request emission-related diagnostic trouble codes detected during current or last completed driving cycle	99
7.8	Service \$08 — Request control of on-board system, test or component.....	100
7.9	Service \$09 — Request vehicle information	104
Annex A (normative) PID (Parameter ID)/OBDMID (On-Board Monitor ID)/TID (Test ID)/INFOTYPE supported definition		114
Annex B (normative) PIDs (Parameter ID) for Services \$01 and \$02 scaling and definition.....		115
Annex C (normative) TIDs (Test ID) scaling description		148
Annex D (normative) OBDMIDs (On-Board Diagnostic Monitor ID) definition for Service \$06		149
Annex E (normative) Unit and Scaling definition for Service \$06		154

py: Institute Of Technology Tallaght, Institute of Technology, Tue Oct 10 10:58:33 E

当前我们解读的 ISO15765-4 协议应用层主要是看目录第七章内容，其中我们的 C300 开发板第一版软件使用到的协议主要是小红色框圈起来部分内容，分别是 Service\$01 当前动力总成诊断数据，Service\$03 排放相关故障码，Service\$09 车辆信息中的车架号，所以小红色框圈起来的内容 C300 开发板也没完全使用完，比如车量信息中只使用了车架号这个协议。第一版软件用到什么内容我们先讲解什么内容。其它协议我们会在后续版本软件中继续解读。

Service\$01 当前动力总成诊断数据（以下简称“当前数据流”），在具体讲解当前数据流的定义前，先了解下什么是 Supported PID。

```

5 //*****文件头*****
6 /*文件: ISO15765_4.c
7 /*作者: Tony.Neulen
8 /*日期: 2011-10-11 10:07
9 /*$产品购买:http://shop115932063.taobao.com
10 //*****包含头文件*****
11 #include "includes.h"
12
13
14
15 //*****系统激活*****
16 CanTxFrameDef EntCmd15765 = {0x7DF, 0x1EDB33F1, CAN_ID_STD, CAN_RTR_DATA, 5, 0x02, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};  

17 //*****读取数据*****
18 CanTxFrameDef DTCmd15765 = {0x7DF, 0x1EDB33F1, CAN_ID_STD, CAN_RTR_DATA, 5, 0x01, 0x03, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};  

19 //*****车架号*****
20 CanTxFrameDef VinCmd15765 = {0x7DF, 0x1EDB33F1, CAN_ID_STD, CAN_RTR_DATA, 5, 0x02, 0x09, 0x02, 0x00, 0x00, 0x00, 0x00};  

21 //*****读取数据*****
22 CanTxFrameDef DSCmd15765 = {0x7DF, 0x1EDB33F1, CAN_ID_STD, CAN_RTR_DATA, 5, 0x02, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00};  

23 //*****尾部*****

```

上图所示，前面讲解中我们提到过 ISO15765-4 协议初始化命令采用 SID（服务字节）是 0x01，PID(参数识别号)是 0x00。这里的 PID=0x00 是作为 Supported PIDs 来使用。Supported PIDs 顾名思义就是识别支持的数据流参数。Supported PIDs 在 C300 开发板第一版软件中并没有具体实现代码，它被定于第二版本软件中使用。这里提前解读，对大家自行阅读协议有比较大的帮助。对比初始化命令 EntCmd15765 和数据流命令 DSCmd15765 的值是完全一样的，其实 DSCmd15765 命令的 PID 就是第三个字节 0x00，是随着请求具体数据流的不同，程序自动对其进行赋值，所以它的 PID 并不是 0x00。但是这两个命令使用相同的 SID=0x01，请求的是相同的服务，即当前数据流。下面我们看下当前数据流是如何请求和响应的。

请求 Supported PIDs，如下图所示。

Table 125 — Request current powertrain diagnostic data request message (read supported PIDs)

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	Request current powertrain diagnostic data request SID	M	01	SIDRQ
#2	PID#1 (PIDs supported: see Annex A)	M	xx	PID
#3	PID#2 (PIDs supported: see Annex A)	U ^a	xx	PID
#4	PID#3 (PIDs supported: see Annex A)	U	xx	PID
#5	PID#4 (PIDs supported: see Annex A)	U	xx	PID
#6	PID#5 (PIDs supported: see Annex A)	U	xx	PID
#7	PID#6 (PIDs supported: see Annex A)	U	xx	PID

^a U = User Optional — PID may be included to avoid multiple PID supported request messages.

读当前数据流之前，可以先读 Supported PIDs，这是诊断仪最标准做法。表格中列出了请求命令的应用层数据。第一字节 0x01 是当前数据流的 SID。第二字节是 PID，但是这个 PID 是用于请求 Supported PIDs。这些 PID 的值有哪些呢？表中告诉我们可以看 Annex A 这个列表。从协议截图出来如下图所示。

ed Copy, Institute Of Technology Tallaght, Institute of Technology, Tue Oct 10 10:58:33 BST 2006, Uncontrolled Copy, (c) Bc

Annex A
(normative)

**PID (Parameter ID)/OBDMID (On-Board Monitor ID)/TID
(Test ID)/INFOTYPE supported definition**

This annex specifies standardized hex values to be used in the request message for Services \$01, \$02, \$05, \$06, \$08, and \$09 to retrieve supported PIDs, OBDMIDs, TIDs, and INFOTYPES.

Table A.1 — Supported PID/OBDMID/TID/INFOTYPE definition				
Supported PID/OBDMID/TID/INFOTYPE (hex)	Scaling/bit Number of data bytes = 4 Data A - D or B - E: bit evaluation		External test equipment SI (Metric) / English display	
00	Data A bit 7 Data A bit 6 : Data D bit 0	01 02 : 20	0 = not supported 1 = supported	ISO 15031-4 specifies the behaviour of the external test equipment for how to interpret the data received to identify supported PIDs/OBDMIDs/TIDs/INFOTYPES for each ECU.
20	Data A bit 7 Data A bit 6 : Data D bit 0	21 22 : 40	0 = not supported 1 = supported	The ECU shall not respond to unsupported PID/TID/MID/InfoTypes ranges unless subsequent ranges have a supported PID(s)/MID(s)/TID(s)/InfoType(s).
40	Data A bit 7 Data A bit 6 : Data D bit 0	41 42 : 60	0 = not supported 1 = supported	
60	Data A bit 7 Data A bit 6 : Data D bit 0	61 62 : 80	0 = not supported 1 = supported	
80	Data A bit 7 Data A bit 6 : Data D bit 0	81 82 : A0	0 = not supported 1 = supported	
A0	Data A bit 7 Data A bit 6 : Data D bit 0	A1 A2 : C0	0 = not supported 1 = supported	
C0	Data A bit 7 Data A bit 6 : Data D bit 0	C1 C2 : E0	0 = not supported 1 = supported	
E0	Data A bit 7 Data A bit 6 : Data D bit 1 Data D bit 0	E1 E2 : FF ISO/SAE reserved (set to 0)	0 = not supported 1 = supported	

Supported PIDs 的 PID 取值范围从 0x00 到 0xE0。这个表格我们暂时只看 PID 取值，其它内容接下来再继续讲。回到 Table125 表格，第三字节往后也是 Supported PIDs 的 PID，但是表格中有一列 Cvt，标注是 M 表示不可或缺的，标注 U 表示用户可选的，从第三字节开始就是用户可选项。U = User Optional — PID may be included to avoid multiple PID supported request messages.

意思就是这些 Supported PIDs 的 PID 可以被包含在命令里以避免太多的 Supported PIDs 的请求信息。根据这个协议的定义可以构建下面两种请求命令。方便起见，我以标准 CAN 为例写出这些命令。

第一种，以 SingleFrame 依次请求命令如下

7DF 8 02 01 00 00 00 00 00 00

7DF 8 02 01 20 00 00 00 00 00

7DF 8 02 01 40 00 00 00 00 00

7DF 8 02 01 60 00 00 00 00 00

7DF 8 02 01 80 00 00 00 00 00

7DF 8 02 01 A0 00 00 00 00 00

7DF 8 02 01 C0 00 00 00 00 00

7DF 8 02 01 E0 00 00 00 00 00

第二种，以 MultipleFrame 传输一次性请求命令如下

7DF 8 10 09 01 00 20 40 60 80

7EO 8 21 A0 C0 E0 00 00 00 00 00 (如果流控帧标识符是 7E8, 此帧标识符是 7EO)

SingleFrame 请求，必须发送一帧 SingleFrame 请求获得一次响应，总共 8 帧请求分别 8 次响应。如果以 MultipleFrame 请求，就会获得一次性响应。关于响应的数据格式和规则如下图所示。

Table 126 — Request current powertrain diagnostic data response message (report supported PIDs)

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	Request current powertrain diagnostic data response SID	M	41	SIDPR
#2	data record of supported PIDs = [1 st supported PID Data A: supported PIDs, Data B: supported PIDs, Data C: supported PIDs, Data D: supported PIDs]	M	xx	PIDREC_ PID
#3		M	xx	DATA_A
#4		M	xx	DATA_B
#5		M	xx	DATA_C
#6		M	xx	DATA_D
:	:	:	:	:
#n-4	data record of supported PIDs = [m th supported PID Data A: supported PIDs, Data B: supported PIDs, Data C: supported PIDs, Data D: supported PIDs]	C1 ^a	xx	PIDREC_ PID
#n-3		C2 ^b	xx	DATA_A
#n-2		C2	xx	DATA_B
#n-1		C2	xx	DATA_C
#n		C2	xx	DATA_D

^a C1 = Conditional — PID value shall be the same value as included in the request message if supported by the ECU.
^b C2 = Conditional — value indicates PIDs supported; range of supported PIDs depends on selected PID value (see C1).

第一字节是当前数据流的响应 SID=0x41, 接下来每 5 个字节对应一个 Supported PIDs 的 PID 的响应。当然如果您是以 SingleFrame 请求获得的响应也是 SingleFrame 的响应。以

MultipleFrame 请求就会获得多组 5 个字节的 Supported PIDs 的 PID 的响应。这 5 个字节的第 1 字节是 Supported PID, 后 4 个字节表示支持的数据流, 怎么表示呢? 看下图的举例。

Table 130 — ECU#1 response: Request current powertrain diagnostic data response message

Message direction:	ECU#1 → External test equipment		
Message Type:	Response		
Data Byte	Description (All PID values are in hexadecimal)	Byte Value (Hex)	Mnemonic
#1	Request current powertrain diagnostic data response SID	41	SIDPR
#2	PID requested	00	PID
#3	Data byte A, representing support for PIDs 01, 03-08	10111111b = \$BF	DATA_A
#4	Data byte B, representing support for PIDs 09, 0B-10	10111111b = \$BF	DATA_B
#5	Data byte C, representing support for PIDs 11, 13, 15	10101000b = \$A8	DATA_C
#6	Data byte D, representing support for PIDs 19, 1C, 20	10010001b = \$91	DATA_D
#7	PID requested	20	PID
#8	Data byte A, representing support for PID 21	10000000b = \$80	DATA_A
#9	Data byte B, representing no support for PIDs 29-30	00000000b = \$00	DATA_B
#10	Data byte C , representing no support for PIDs 31-38	00000000b = \$00	DATA_C
#11	Data byte D, representing no support for PIDs 39-40	00000000b = \$00	DATA_D

表中 PID=0x00 响应的数据第三字节为 0xBF, 转换为二进制是 10111111, 在 Annex A 表格中定义 1 表示支持, 0 表示不支持, 而表示的 PID 值按照从高字节到低字节, 从高位到低位表示 PID=0x01, 0x02, 0x03, 0x04, 0x05, 0x06. 第三字节的 10111111 分别表示 PID=0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08。而支持与否的结果是 PID=0x01, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08 支持, 但是 PID=2 不支持。第四字节一样是 0xBF, 这时候 PID=0x09, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x10 支持, 但是 PID=0xA 不支持。以此类推剩下两个字节表示 PID=0x11 到 0x20 是否支持的情况。接下来是 PID=0x20 响应的 Supported PIDs 的情况。从而可以知道一个 Supported PIDs 的 PID 的响应由四个字节表示, 每一个比特位表示一个 PID 是否支持的情况, 也就是一个 Supported PIDs 的 PID 的响应可以获知 32 个 PID 是否支持的情况。上面的表格如果是实际通信的数据是怎么样的呢? 我们下面还原下帮助大家整体理解, 以标准 CAN, 汽车由 ECU#1 响应为例。

7DF 8 03 01 00 20 00 00 00 00 <---这是 C300 开发板发送的请求

7E8 8 10 00 41 00 BF BF A8 91 <---这是汽车 ECU#1 响应

7E0 8 30 00 00 00 00 00 00 00 <---这是 C300 开发板发出的流控帧

7E8 8 21 20 80 00 00 00 00 00 <---这是汽车 ECU#1 响应

在读取 PID 支持列表后, 就可以按照支持的情况直接读取当前系统支持的数据流了。这里以发动机转速和车速为例告诉大家如何读取这两个数据流数值。首先在 Annex B 中找到发动机转速和车速的定义。如下图所示。

Table B.13 — PID \$0C definition

PID (hex)	Description	Data byte	Min. value	Max. Value	Scaling/bit	External test equipment SI (Metric) / English display
0C	Engine RPM	A, B	0 min ⁻¹	16383,75 min ⁻¹	1/4 rpm per bit	RPM: xxxx min ⁻¹
Engine RPM shall display revolutions per minute of the engine crankshaft.						

Table B.14 — PID \$0D definition

PID (hex)	Description	Data byte	Min. value	Max. Value	Scaling/bit	External test equipment SI (Metric) / English display
0D	Vehicle Speed Sensor	A	0 km/h	255 km/h	1 km/h per bit	VSS: xxx km/h (xxx mph)
VSS shall display vehicle road speed, if utilized by the control module strategy. Vehicle speed may be derived from a vehicle speed sensor, calculated by the PCM using other speed sensors, or obtained from the vehicle serial data communication bus.						

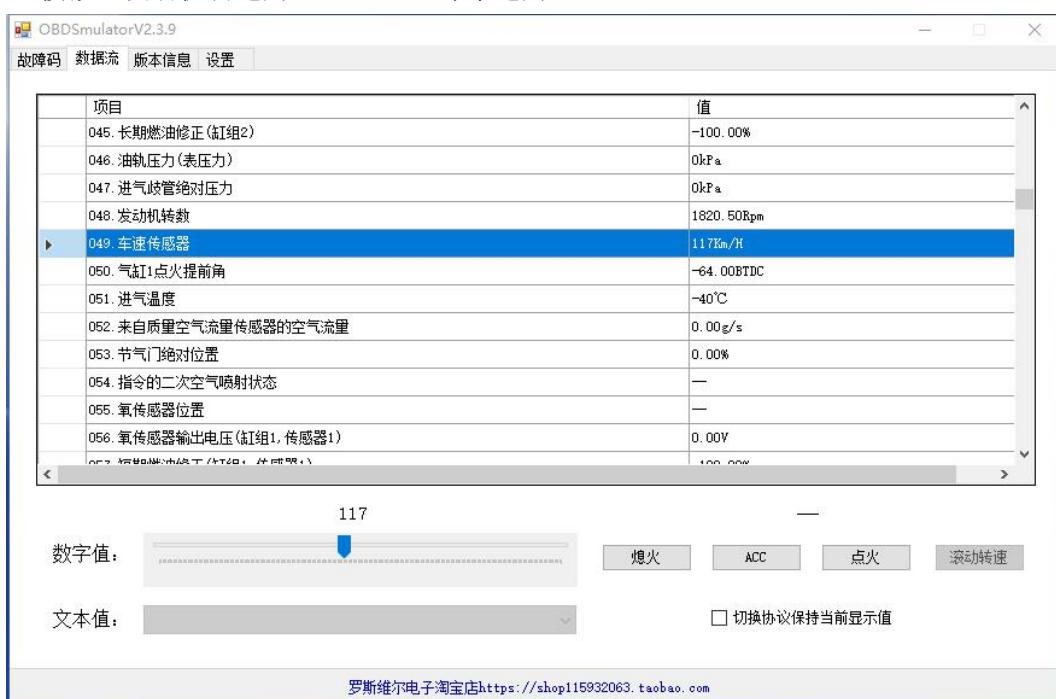
它们的 PID 分别是 0x0C 和 0x0D, 这时候就可以构建它们的请求命令。以标准 CAN 为例。

发动机转速请求命令: 7DF 8 02 01 0C 00 00 00 00 00

车速请求命令: 7DF 8 02 01 0D 00 00 00 00 00

下面我们从 C300 开发板和模拟器通信数据看下响应数据是否与上面表格定义一致。

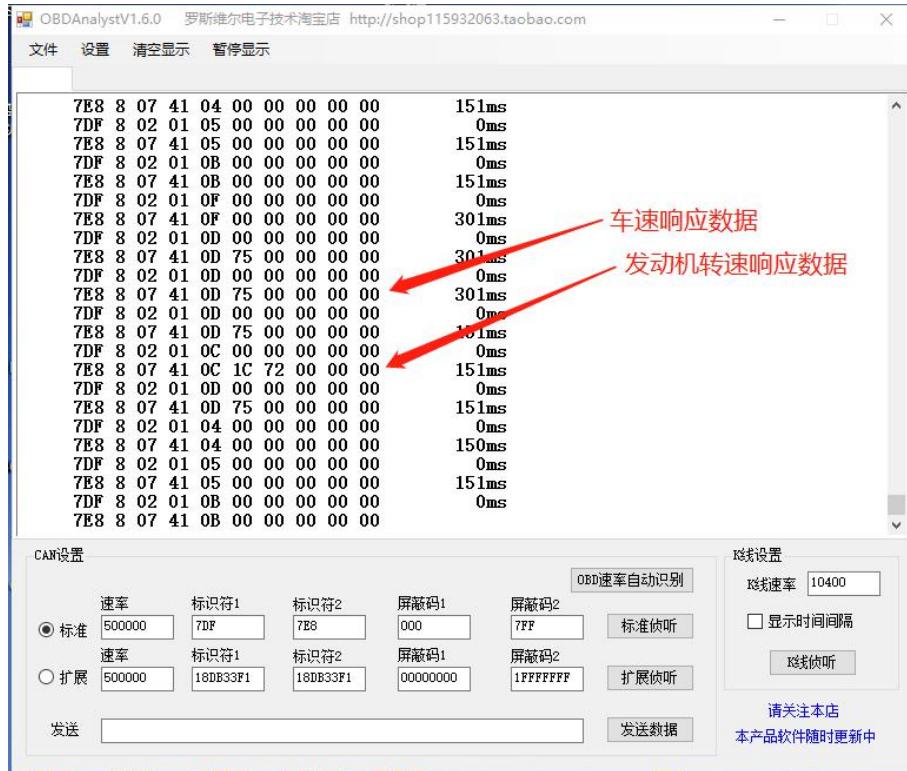
设置模拟器发动机转速为 1820.50 RPM, 车速为 117KM/H.



C300 开发板服务器网站显示值如下图所示，与模拟器模拟值一致。



OBD 分析仪采集到的通信数据如下图所示。



发动机转速响应数据 7E8 8 07 41 0C 1C 72 00 00 00

数据中应用层数据为 41 0C 1C 72, 41 是响应 SID, 0C 是发动机转速 PID, 转速值就是 1C72. 按照上面 TableB.13 表格定义换算 1/4rmp/bit, 且 DataByte 值是 AB, 意思是将两个字节取值除以 4 即可得到发动机转速。将 0x1C72 转换成十进制是 7282, $7282/4=1820.5$. 与 C300 读到的值一致。

车速响应数据 7E8 8 07 41 0D 75 00 00 00 00

数据中应用层数据为 41 0D 75, 41 是响应 SID, 0D 是车速 PID, 车速值就是 75. 按照上面 TableB.14 表格定义换算 1km/h per bit, DataByte 值是 A, 意思就是一个字节取值乘以 1, 其实就是 0x75 十进制的值 117.

Service\$03 与排放相关的诊断故障码（以下简称“故障码”）。首先看下故障码结构，如下图所示。

DTC High Byte									DTC Low Byte								
\$9				\$2					\$3				\$4				
1	0	0	1	0	0	1	0		0	0	1	1	0	1	0	0	
B	1			2					3				4				

Figure 9 — Example of 2-byte diagnostic trouble code structure

故障码通常由两个字节组成，当然有些协议是三个字节组成。对于法规排放协议 ISO15765-4 定义是两个字节表示一个故障码。高字节高两位表示故障位置划分，可表示车身，底盘，动力总成和网络。如下图所示。

Table 1 — General code specifications

System	Code categories	Hex value	Appendix
Body	B0xxx – B3xxx	8xxx – Bxxx	B
Chassis	C0xxx – C3xxx	4xxx – 7xxx	C
Powertrain	P0xxx – P3xxx	0xxx – 3xxx	P
Network	U0xxx – U3xxx	Cxxx – Fxxx	U

故障码高两位是 00, 表示动力总成 Powertrain, 用字母 P 表示;

故障码高两位是 01, 表示底盘 Chassis, 用字母 C 表示;

故障码高两位是 10, 表示车身 Body, 用字母 B 表示;

故障码高两位是 11, 表示网络 Network, 用字母 U 表示;

剩下的高字节低 6 位和低字节 8 位总共 14 位构成了故障值。整个故障编码再法规协议中的故障定义可参考 ISO15031-6 协议的 Annex B 中的定义。如下图所示

Annex B (normative)

Powertrain system diagnostic trouble codes

B.1 P00XX Fuel and air metering and auxiliary emission controls

Table B.1 — P00XX Fuel and air metering and auxiliary emission controls

DTC number	DTC naming	Location
P0000	ISO/SAE reserved	
P0001	Fuel Volume Regulator Control Circuit/Open	
P0002	Fuel Volume Regulator Control Circuit Range/Performance	
P0003	Fuel Volume Regulator Control Circuit Low	
P0004	Fuel Volume Regulator Control Circuit High	
P0005	Fuel Shutoff Valve "A" Control Circuit/Open	
P0006	Fuel Shutoff Valve "A" Control Circuit Low	
P0007	Fuel Shutoff Valve "A" Control Circuit High	
P0008	Engine Position System Performance	Bank 1
P0009	Engine Position System Performance	Bank 2
P000A	"A" Camshaft Position Slow Response	Bank 1
P000B	"B" Camshaft Position Slow Response	Bank 1
P000C	"A" Camshaft Position Slow Response	Bank 2

比如 P0001 故障码, 表示动力总成故障, 燃油调节器控制电路/开路。那么故障码是如何请求和获得汽车响应呢? 相关定义可参阅 ISO15031-5 协议, 如下图所示。

Table 145 — Request emission-related DTC request message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	Request emission-related DTC request SID	M	03	SIDRQ

Table145 告诉我们故障码请求命令只需要在数据中承载故障码的 SID=0x03 即可。下图展示了 C300 开发板中读故障码的命令。

```

/*系统激活*/
CanTxFrameDef EntCmd15765 = {0x7DF, 0x19DB33F1, CAN_ID_STD, CAN_RTR_DATA, 9, 0x02, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};

/*读故障码*/
CanTxFrameDef DTCmd15765 = {0x7DF, 0x19DB33F1, CAN_ID_STD, CAN_RTR_DATA, 9, 0x01, 0x03, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};

/*读数据流*/
CanTxFrameDef VinCmd15765 = {0x7DF, 0x19DB33F1, CAN_ID_STD, CAN_RTR_DATA, 9, 0x02, 0x09, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00};

/*读车架号*/
CanTxFrameDef DS Cmd15765 = {0x7DF, 0x19DB33F1, CAN_ID_STD, CAN_RTR_DATA, 9, 0x02, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};

```

读故障码命令,
SID=0x03

故障码请求命令向汽车发送后，如果有故障码会得到故障码响应数据格式如下所示。

Table 146 — Request emission-related DTC response message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	Request emission-related DTC response SID	M	43	SIDPR
#2	# of DTC = [no emission-related DTCs stored emission-related DTCs stored]	M	xx =[00, 01 - FF	#OFDTC
#3	DTC#1 (High Byte)	C ^a	xx	DTC1HI
#4	DTC#1 (Low Byte)	C	xx	DTC1LO
:	:	:	xx	
#n-1	DTC#n (High Byte)	C	xx	DTCmHI
#n	DTC#n (Low Byte)	C	xx	DTCmLO

^a C = Conditional — DTC#1 - DTC#m are only included if # of DTC parameter value ≠ \$00.

第一字节是响应故障码的 SID. 第二字节是故障码个数。接下来每两个字节表示一个故障码。为了更好的理解，ISO15031-5 协议进行了举例，如下图所示。

Table 147 — Request emission-related diagnostic trouble codes request message

Message direction:	External test equipment → All ECUs		
Message Type:	Request		
Data Byte	Description (all values are in hexadecimal)	Byte Value (Hex)	Mnemonic
#1	Request emission-related DTCs request SID	03	SIDRQ

Table 148 — Request emission-related diagnostic trouble codes response message

Message direction:	ECU #1 → External test equipment		
Message Type:	Response		
Data Byte	Description (all values are in hexadecimal)	Byte Value (Hex)	Mnemonic
#1	Request emission-related DTCs response SID	43	SIDPR
#2	# of DTC (number of emission-related DTCs stored in this ECU)	06	#OFDTC
#3	DTC High Byte of P0143	01	DTC1HI
#4	DTC Low Byte of P0143	43	DTC1LO
#5	DTC High Byte of P0196	01	DTC2HI
#6	DTC Low Byte of P0196	96	DTC2LO
#7	DTC High Byte of P0234	02	DTC3HI
#8	DTC Low Byte of P0234	34	DTC3LO
#9	DTC High Byte of P02CD	02	DTC4HI
#10	DTC Low Byte of P02CD	CD	DTC4LO
#11	DTC High Byte of P0357	03	DTC5HI
#12	DTC Low Byte of P0357	57	DTC5LO
#13	DTC High Byte of P0A24	0A	DTC6HI
#14	DTC Low Byte of P0A24	24	DTC6LO

为了客观理解上图表格，我们直接使用 OBD 模拟器+C300 开发板+OBD 分析仪进行模拟 Table147 和 Table148 所讲述的内容。

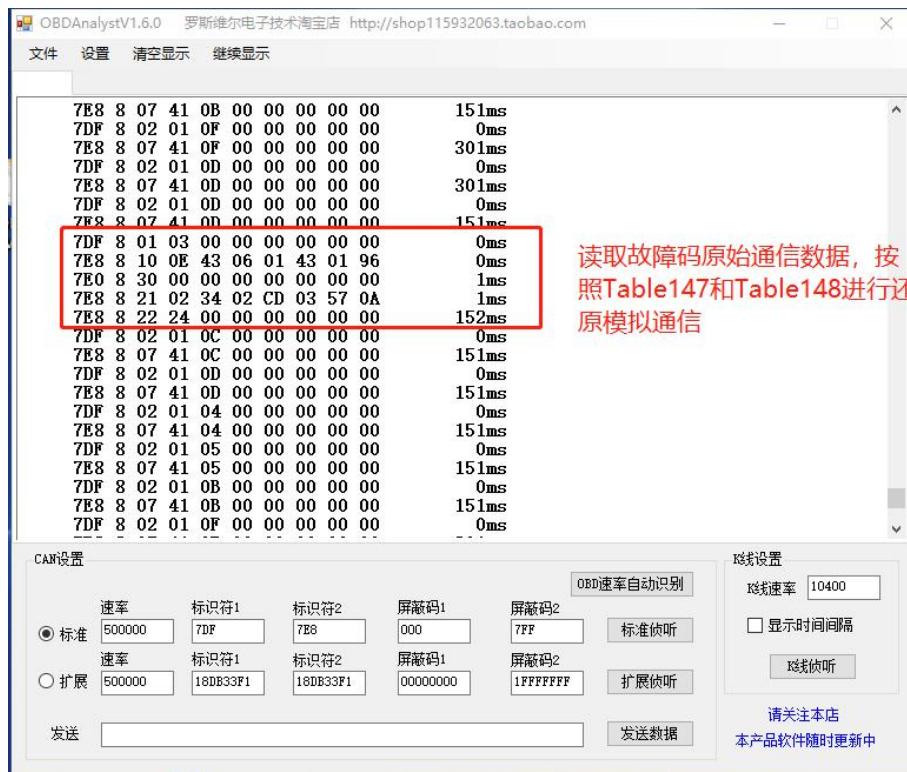
首先使用 OBD 模拟器模拟 Table148 的数据，如下图所示，在 OBD 模拟器上设置故障码 P0143, P0196, P0234, P02CD, P0357, P0A24。



然后，在C300开发板上传的服务器网站上读取故障码如下图所示，与模拟器设置值完全一致。



最后，我们看下，C300开发板与模拟器通信数据，如下图所示OBD分析仪采集到的数据。



这里具体分析数据，按照 Table147 发送故障码请求信息 SID=0x03，承载在 SingleFrame 上，以标准 CAN 功能性标识符 7DF 发送（此处以标准 CAN 为例，扩展 CAN 类似把对应标识符替换即可），请求故障码信息标准 CAN 数据帧位 7DF 8 01 03 00 00 00 00 00 00。然后获得 OBD 模拟器故障码响应数据。

7E8 8 10 0E 43 06 01 43 01 96

7E8 8 21 02 34 02 CD 03 57 0A

7E8 8 22 24 00 00 00 00 00 00

流控帧是 C300 开发板发送的，不属于响应数据，这里就省略了。按照 MultipleFrame 传输规则，把响应数据提取出应用层原始数据为 06 01 43 01 96 02 34 02 CD 03 57 0A 24。这时候发现这组数据与 Table148 表格完全一致。06 表示故障码个数，总共读到 6 个故障码。接下来两个字节表示一个故障码，01 43 表示故障码 P0143, 01 96 表示故障码 P0196, 依次类推，数据表示 6 个故障码分别为 P0143, P0196, P0234, P02CD, P0357, P0A24。上面的模拟过程完全正确。这里的 6 个故障码它们的高字节的高两位均是二进制 00，所以代表动力总成故障，所有故障码要以 P 开头。

Service\$09 请求车辆信息，这里我们只讨论当前 C300 开发板第一版软件用到的车架号信息，其它车辆信息待后续版本再继续讲解。

请求车辆信息和数据流一样，有一个类似于 Supported PID 的 Supported InfoType，叫被支持的信息类型。获取支持的列表和数据流一样，只是 SID=0x09。如下图所示。

Table 181 — Request vehicle information request message (request supported InfoType)

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	Request vehicle information request SID	M	09	SIDRQ
#2	InfoType#1 (InfoTypes supported: Annex A)	M	xx	INF_TYP
#3	InfoType#2 (InfoTypes supported: Annex A)	U ^a	xx	INF_TYP
#4	InfoType#3 (InfoTypes supported: Annex A)	U	xx	INF_TYP
#5	InfoType#4 (InfoTypes supported: Annex A)	U	xx	INF_TYP
#6	InfoType#5 (InfoTypes supported: Annex A)	U	xx	INF_TYP
#7	InfoType#6 (InfoTypes supported: Annex A)	U	xx	INF_TYP

^a U = User Optional — InfoType may be included to avoid multiple InfoType supported request messages.

这里我举例如何利用标准 CAN 请求信息查询是否支持车架号信息。按照上表构建 ISO15765-4 协议命令是 7DF 8 02 09 00 00 00 00 00 00。响应数据格式如下面所示。

Table 182 — Request vehicle information response message (report supported InfoType)

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	Request vehicle information response SID	M	49	SIDPR
#2	data record of supported InfoTypes = [1 st supported InfoType Data A: supported InfoTypes, Data B: supported InfoTypes, Data C: supported InfoTypes, Data D: supported InfoTypes]	M	xx	INF_TYPREC
#3		M	xx	INF_TYP
#4		M	xx	DATA_A
#5		M	xx	DATA_B
#6		M	xx	DATA_C
:	:	:	:	:
#n-4	data record of supported InfoTypes = [m th supported InfoType Data A: supported InfoTypes, Data B: supported InfoTypes, Data C: supported InfoTypes, Data D: supported InfoTypes]	C1 ^a	xx	INF_TYPREC
#n-3		C2 ^b	xx	INF_TYP
#n-2		C2	xx	DATA_A
#n-1		C2	xx	DATA_B
#n		C2	xx	DATA_C
				DATA_D

^a C1 = Conditional — INFOTYPE value shall be the same value as included in the request message if supported by the ECU.

^b C2 = Conditional — Value indicates INFOTYPES supported; range of supported INFOTYPES depends on selected INFOTYPE value (see C1).

仔细观察该表发现，其实除了响应 SID 的值和数据的命名与数据流的 Supported PID 的响应数据不一样外，格式是完全一样的，原理也完全一样。所以只要汽车支持 ISO15765-4 11BIT

发送命令 7DF 8 02 09 00 00 00 00 00 00 或者汽车支持 ISO15765-4 29BIT 发送命令 18DB33F1 8 02 09 00 00 00 00 00 00 00 , 获得的响应数据信息的第三字节第二位是 1, 就说明该车支持车架号读取。为什么是第二位呢? 因为 InfoType=0x02 表示车架号的信息类型。如下图所示。

Table 185 — Request vehicle information request message

Message direction:	External test equipment → All ECUs		
Message Type:	Request		
Data Byte	Description (all values are in hexadecimal)	Byte Value (Hex)	Mnemonic
#1	Request vehicle information request SID	09	SIDRQ
#2	InfoType: 02 - VIN (Vehicle Identification Number)	02	INFTYP

Table 186 — Request vehicle information response message

Message direction:	ECU #1 → External test equipment		
Message Type:	Response		
Data Byte	Description (all values are in hexadecimal)	Byte Value (Hex)	Mnemonic
#1	Request vehicle information response SID	49	SIDPR
#2	InfoType: 02 - VIN (Vehicle Information Number)	02	INFTYP
#3	Number of data items: 01	01	NODI
#4	1 st ASCII character of VIN: '1'	31	VIN
#5	2 nd ASCII character of VIN: 'G'	47	VIN
#6	3 rd ASCII character of VIN: '1'	31	VIN
#7	4 th ASCII character of VIN: 'J'	4A	VIN
#8	5 th ASCII character of VIN: 'C'	43	VIN
#9	6 th ASCII character of VIN: '5'	35	VIN
#10	7 th ASCII character of VIN: '4'	34	VIN
#11	8 th ASCII character of VIN: '4'	34	VIN
#12	9 th ASCII character of VIN: '4'	34	VIN
#13	10 th ASCII character of VIN: 'R'	52	VIN
#14	11 th ASCII character of VIN: '7'	37	VIN
#15	12 th ASCII character of VIN: '2'	32	VIN
#16	13 th ASCII character of VIN: '5'	35	VIN
#17	14 th ASCII character of VIN: '2'	32	VIN
#18	15 th ASCII character of VIN: '3'	33	VIN
#19	16 th ASCII character of VIN: '6'	36	VIN
#20	17 th ASCII character of VIN: '7'	37	VIN

根据上表 Table185 和 Table186, 我们还原整个请求和响应车架号的过程。以标准 CAN 为例, 如下所示。

Table186 响应的车架号为 1G1JC5444R7252367

还原数据过程:

7DF 8 02 09 02 00 00 00 00 <-这是 C300 开发板发送的车架号请求
 7E8 8 10 14 49 02 01 31 47 31 <-汽车响应 FirstFrame, 承载三个字节车架号信息
 7E0 8 30 00 00 00 00 00 00 <-这是 C300 开发板发送流控帧命令
 7E8 8 21 4A 43 35 34 34 34 52 <-汽车响应 ConsecutiveFrame, 承载 7 个字节车架号信息
 7E8 8 22 37 32 35 32 33 36 37 <-汽车响应 ConsecutiveFrame, 承载 7 个字节车架号信息

从响应数据中提取应用层数据为: 49 02 01 31 47 31 4A 43 35 34 34 34 52 37 32 35 32 33 36 37

第一字节是车架号响应 SID=0x49, 第二字节是车架号 InfoType=0x02, 第三字节是数据个数, 因为车架号每辆车只有 1 个, 所以是 0x01, 接下来的 17 个字节表示车架号 ASCII 编码。

7.2.2 ISO15765-4 协议在 Neulen TBOX C300 开发板中的程序实现

前面的 SAEJ1939 协议讲解的时候我们提到函数 NL_OBD_SendCANFrame, 如下图所示。

```

23 //*****
24 * @描述: ISO15765唤醒判断
25 * @参数: NONE
26 * @返回值: NL_OK:支持 NL_NOK:不支持
27 *****/
28 NLStatus ISO15765_4_WakeUp(uint8_t s, PROTypeDef pro)
29 {
30     NL_Status err;
31     if(pro == ISO15765_4STD_500K)
32     {
33         NL_OBD_SelectCAN(CAN_6_14);
34         NL_OBD_CANConfig(s,CAN_ID_STD,0x7e0,0x7e0,0x7f0,0x7f0);
35         NL_OBD_SendCANFrame(pro,&EntCmd15765,800,&err);
36     }
37     else
38     {
39         NL_OBD_SelectCAN(CAN_6_14);
40         NL_OBD_CANConfig(s,CAN_ID_EXT,0x18DA0000,0x18DA0000,0x1FFF0000,0x1FFF0000);
41         NL_OBD_SendCANFrame(pro,&EntCmd15765,800,&err);
42     }
43     _NL_Delay(150);
44     return err;
45 }
```

这个函数只要是基于 CAN 总线通信的诊断协议, 都被用来实现应用层协议以下的网络层协议, 数据链路层协议等。函数的第一个参数用来表示当前执行的诊断协议, 这里可以填写 SAEJ1939, ISO15765_4STD_500K 和 ISO15765_4EXT_500K。上图中的程序片段有两个 NL_OBD_SendCANFrame 函数, 第一个参数由 pro 赋值, 这两个函数 pro 的值分别为 ISO15765_4STD_500K 和 ISO15765_4EXT_500K, 也就是使用这两个协议分别发送 EntCmd15765 命令。ISO15765-4 协议除了定义 500K 传输速率外, 还定义了 250K 速率, 250K 速率的 ISO15765-4 在实车测试中并没有发现, 所以代码中我们仅讨论 ISO15765_4STD_500K 和 ISO15765_4EXT_500K 两种情况。

为了演示 NL_OBD_SendCANFrame 函数功能, 根据 ISO15031-5 协议中 Table148 进行故障码模拟, 看 NL_OBD_SendCANFrame 函数如何进行数据处理, 并提取应用层原始数据。

Table 148 — Request emission-related diagnostic trouble codes response message

Message direction:	ECU #1 → External test equipment		
Message Type:	Response		
Data Byte	Description (all values are in hexadecimal)	Byte Value (Hex)	Mnemonic
#1	Request emission-related DTCs response SID	43	SIDPR
#2	# of DTC {number of emission-related DTCs stored in this ECU}	06	#OFDT
#3	DTC High Byte of P0143	01	DTC1HI
#4	DTC Low Byte of P0143	43	DTC1LO
#5	DTC High Byte of P0196	01	DTC2HI
#6	DTC Low Byte of P0196	96	DTC2LO
#7	DTC High Byte of P0234	02	DTC3HI
#8	DTC Low Byte of P0234	34	DTC3LO
#9	DTC High Byte of P02CD	02	DTC4HI
#10	DTC Low Byte of P02CD	CD	DTC4LO
#11	DTC High Byte of P0357	03	DTC5HI
#12	DTC Low Byte of P0357	57	DTC5LO
#13	DTC High Byte of P0A24	0A	DTC6HI
#14	DTC Low Byte of P0A24	24	DTC6LO

模拟故障码 P0143, 看 NL_OBD_SendCANFrame 函数对 SingleFrame 的处理。

1. 设置模拟器协议 ISO15765-4 11BIT 500K，并设置故障码 P0143。



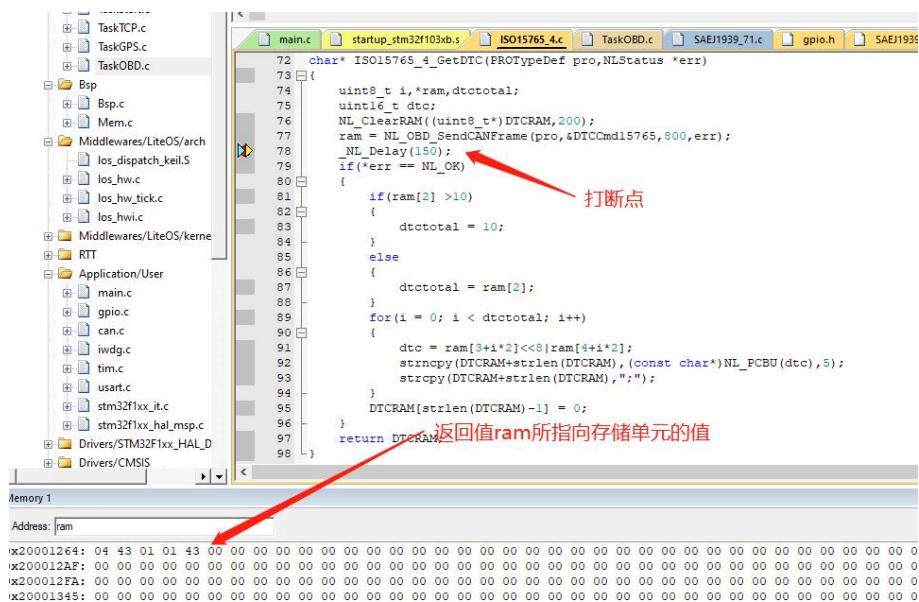
2. C300 开发板读到的结果。



3. OBD 分析仪采集到的读故障码原始 CAN 数据。



4. 在 C300 开发板工程代码中打断点，查看 NL_OBD_SendCANFrame 函数返回值指向的存储单元，如下所示。



代码中 ram 所指向存储单元的值为 04 43 01 01 43。其中 43 01 01 43 就是我们要取的应用层原始数据。第一字节 04 是我加进去用以表示有效字节的长度。所以 NL_OBD_SendCANFrame 函数可以正确处理 ISO15765-4 标准 CAN 的 SingleFrame 通信，而 ISO15765-4 扩展 CAN 的 SingleFrame 通信如下所示。

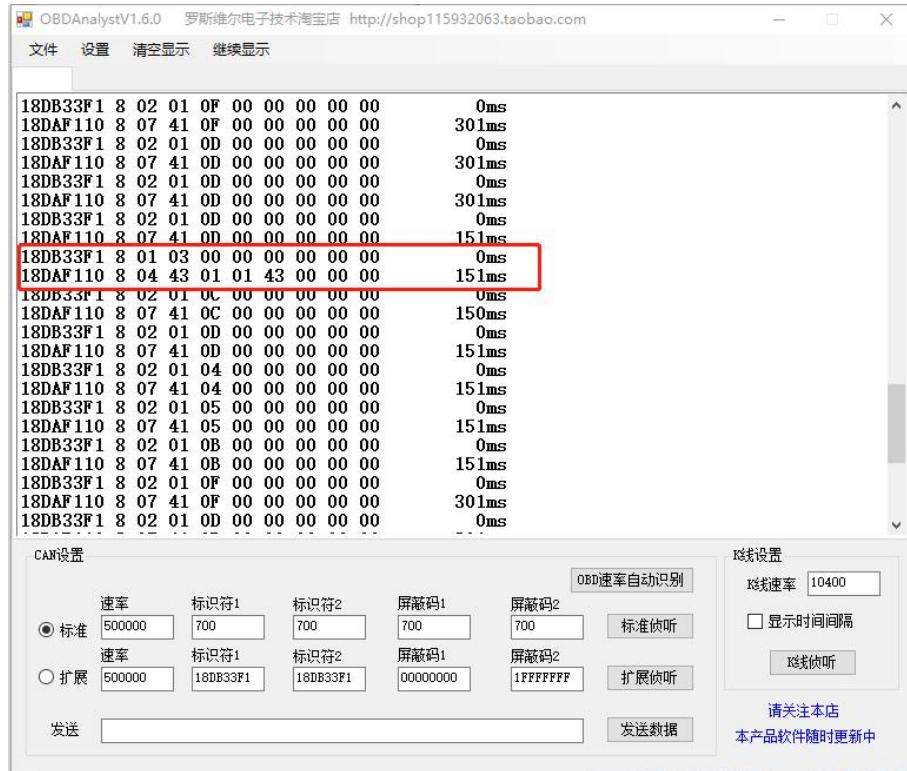
1. 设置模拟器协议 ISO15765-4 29BIT 500K，并设置故障码 P0143。



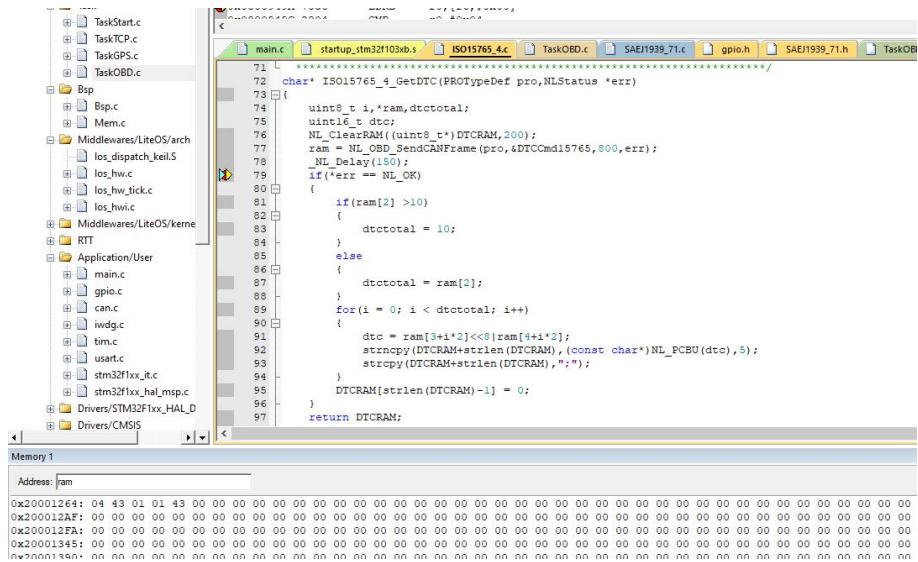
2. C300 开发板读到的结果。

东经	11623.48000	北纬	3954.43813
协议	ISO15765-4(29bit ID 500K)	VIN	---
ISO15031-5乘用车数据流 (可任意定义)			
故障码	P0143		
车速	计算负荷值	发动机冷却液温度	进气岐管绝对压力

3. OBD 分析仪采集到的读故障码原始 CAN 数据。这时候只是 CAN 标识符和 11BIT 标准 CAN 不一样，数据域是一样的。



4. 在 C300 开发板工程代码中打断点，查看 NL_OBD_SendCANFrame 函数返回值指向的存储单元的值和 11BIT 标准 CAN 是一样的。



所以 NL_OBD_SendCANFrame 函数可以顺利处理 ISO15765-4 11BIT 和 29BIT 的 SingleFrame 数据，并能提取应用层原始数据作为返回值。

模拟故障码 P0143, P0196, P0234, P02CD, P0357, P0A24, 看 NL_OBD_SendCANFrame 函数对 MultipleFrame 的处理。

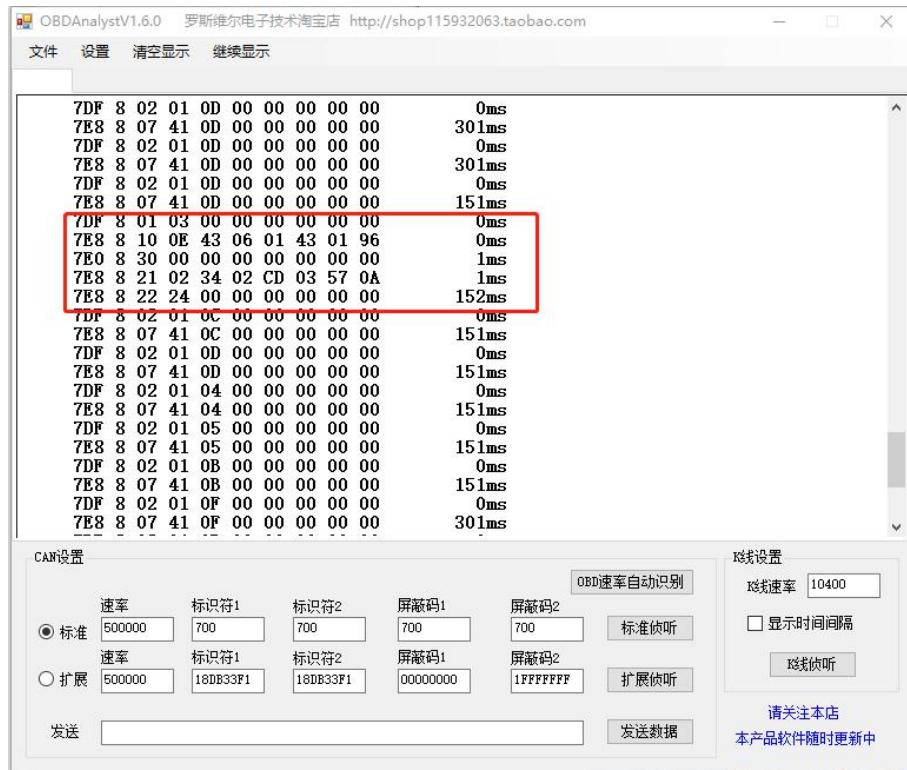
1. 设置模拟器协议 ISO15765-4 11BIT 500K, 并设置故障码 P0143, P0196, P0234, P02CD, P0357, P0A24。



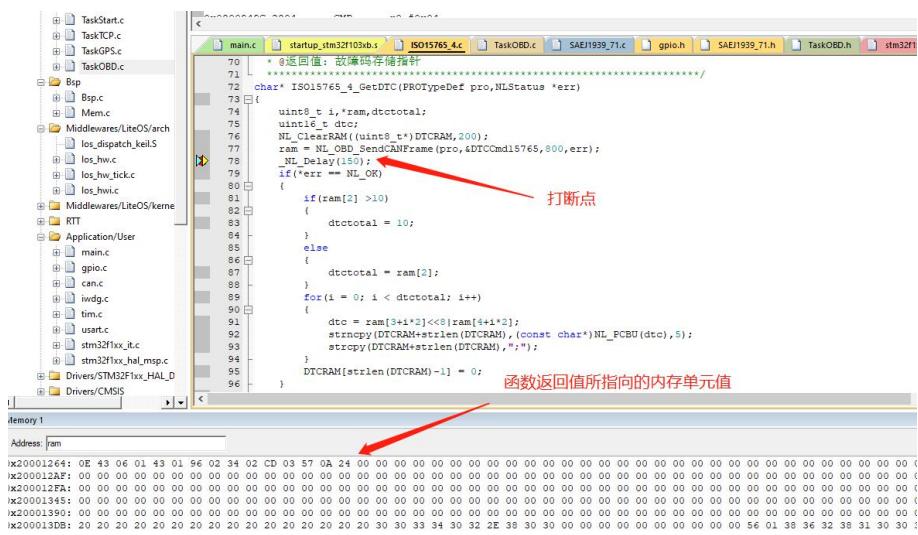
2. C300 开发板读到的结果。

东经	11623.48000	北纬	3954.43813
协议	ISO15765-4(11bit ID 500K)	VIN	---
ISO15031-5乘用车数据流 (可任意定义)			
故障码		P0143;P0196;P0234;P02cd;P0357;P0a24	
车速	计算负荷值	发动机冷却液温度	进气歧管绝对压力

3. OBD 分析仪采集到的请求和响应 MultipleFrame 数据。



4. 在 C300 开发板工程代码中打断点，查看 NL_OBD_SendCANFrame 函数返回值。



函数返回值所指向的存储单元的值是 0E 43 06 01 43 01 96 02 34 02 CD 03 57 0A 24。

第1字节是有效字节长度，这是我编写函数定义的，当前表示有 14 个字节用于应用层数据。

第2字节是故障码响应 SID。

第3字节是故障码个数，06 表示有 6 个故障码。

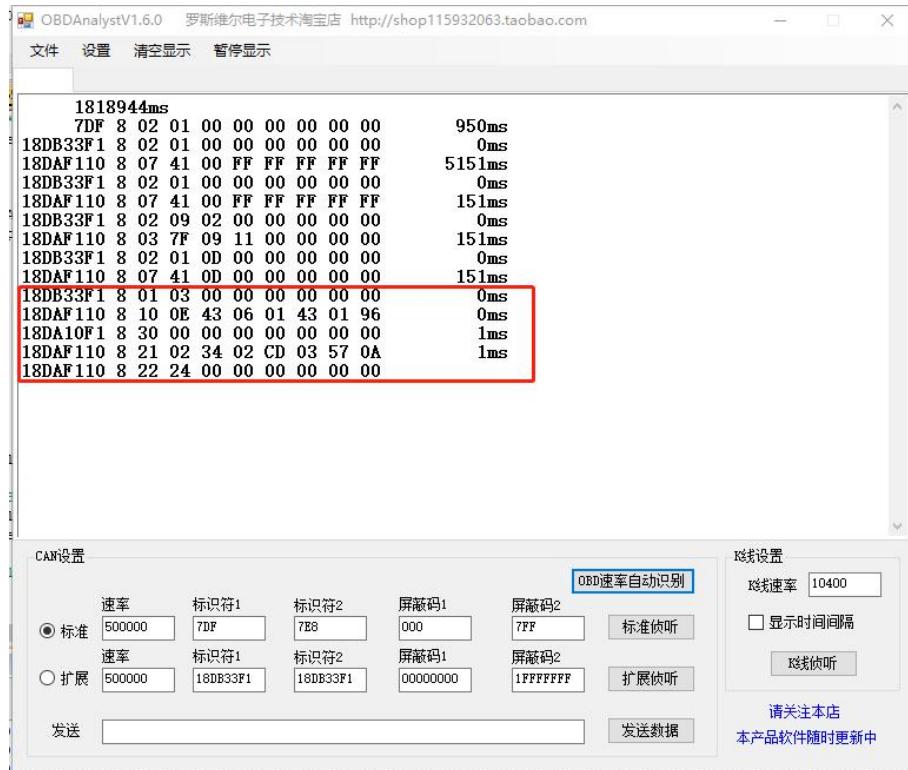
剩下的 12 个字节表示故障码，每两个字节组成一个故障码。

从第 2 字节开始的数据都是应用层的原始数据，因为当前模拟的故障码是按照 ISO15031-5 协议 Table148 模拟的，所以函数返回值所指向的存储单元的值 0E 43 06 01 43 01 96 02 34 02 CD 03 57 0A 24 与下图 Table148 红色框的值完全一致。

Table 148 — Request emission-related diagnostic trouble codes response message

Message direction:	ECU #1 → External test equipment		
Message Type:	Response		
Data Byte	Description (all values are in hexadecimal)	Byte Value (Hex)	Mnemonic
#1	Request emission-related DTCs response SID	43	SIDPR
#2	# of DTC {number of emission-related DTCs stored in this ECU}	06	#OFDTCTC
#3	DTC High Byte of P0143	01	DTC1HI
#4	DTC Low Byte of P0143	43	DTC1LO
#5	DTC High Byte of P0196	01	DTC2HI
#6	DTC Low Byte of P0196	96	DTC2LO
#7	DTC High Byte of P0234	02	DTC3HI
#8	DTC Low Byte of P0234	34	DTC3LO
#9	DTC High Byte of P02CD	02	DTC4HI
#10	DTC Low Byte of P02CD	CD	DTC4LO
#11	DTC High Byte of P0357	03	DTC5HI
#12	DTC Low Byte of P0357	57	DTC5LO
#13	DTC High Byte of P0A24	0A	DTC6HI
#14	DTC Low Byte of P0A24	24	DTC6LO

至于 ISO15765-4 29BIT 500K 的扩展 CAN 协议，只是标识符的区别，NL_OBD_SendCANFrame 函数处理效果如下图所示。



下面我们具体看 `NL_OBD_SendCANFrame` 函数关于 ISO15765-4 协议这部分代码的实现。如下图所示。

NL_OBD_SendCANFrame 函数有 4 个参数：

第 1 个参数 pro，当前要处理的 CAN 协议类型，协议类型可以是 SAEJ1939，

ISO15765_4STD_500K, ISO15765_4EXT_500K。当前我们讨论的是后两种协议。

第 2 个参数 TxMessage, 待发送的请求数据，如上图种的 EntCmd15765, DTCCmd15765, VinCmd15765, DSCmd15765。

第 3 个参数 TimeOut, 等待响应超时时间，单位为毫秒。如果在该设置时间内没有得到汽车或者模拟器响应请求数据，则超时。超时标志由第 4 个参数表示。

第 4 个参数 *err, 该参数用以表示函数请求响应是否超时，参数必须为 NLStatus 定义变量的地址。超时则未得到汽车或者模拟器响应 *err = NL_NOK; 没有超时则说明在第 3 个参数规定的时间内获得了汽车或者模拟器针对当前请求信息的响应 *err=NL_OK。

看函数体本身代码，51 行将第 1 个参数值赋给全局变量 OBDFlag.ProType，它将在中断中起作用。52 行到 56 行用于 SAEJ1939 协议，前面已经解释过，这里不再讨论。57 行到 64 行通过判断 pro 的值确定当前请求数据是以 11bit 标准 CAN 请求还是以 29bit 扩展 CAN 请求，并设置发送数据的 IDE 类型。65 行 OBDFlag.RxFlag 用来表示当前协议请求数据发送后，中断是否获得完整响应数据，如果获得完整响应数据 OBDFlag.RxFlag=NL_SUCCESS，否则 OBDFlag.RxFlag=NL_FAILURE，此处让 OBDFlag.RxFlag=NL_FAILURE 就是为了等待中断赋值 NL_SUCCESS 给 OBDFlag.RxFlag，如果中断始终没有获得请求数据的完整汽车响应数据 OBDFlag.RxFlag 的值一直是 NL_FAILURE。66 行 *err 首先赋值 NL_NOK，如果在第 3 个参数规定时间内获得请求数据的汽车完整响应数据，OBDFlag.RxFlag=NL_SUCCESS，我们就让 *err=NL_OK，下面的 68 行到 77 行代码就是实现这个内容的。67 行代码通过 _NL_OBD_CAN_Transmit(TxMessage) 函数把第 2 个参数待发送的请求数据发送出去。68 行到 77 行代码刚刚提到过，通过 for 循环延时等待，等待中断获得请求数据的汽车完整响应数据，通过 OBDFlag.RxFlag=NL_SUCCESS 判断中断的处理结果，如果规定时间内 OBDFlag.RxFlag=NL_SUCCESS 就让 *err=NL_OK，否则保持不变 *err=NL_NOK。这里每次循环通过系统延时函数 _NL_Delay(1) 限定每次循环时间为 1 毫秒。循环次数由函数第 3 个参数 TimeOut 决定，所以赋值多少就是多少毫秒的等待，除非 OBDFlag.RxFlag=NL_SUCCESS 则立即退出循环等待。78 行返回响应数据的存储地址。

从 NL_OBD_SendCANFrame 函数源码，我们只看到了请求数据的发送和等待，并没有看到响应数据的处理和存储，这部分的实现主要是通过 CAN 中断完成。CAN 中断和

NL_OBD_SendCANFrame 函数体代码是通过全局变量 OBDFlag.RxFlag 保持的同步。下面我们具体看 CAN 中断源码如何处理响应数据的。下图所示。

```

405 /* **** */
406 * @描述: CAN信息挂起回调函数
407 * @参数: CAN_HandleTypeDef *CanHandle
408 * @返回值: void
409 ****/
410 ITStatus MUXFrameFlagBool = RESET;
411 uint8_t MUXFrameCount;
412 uint8_t MUXFrame.MaxValue;
413 TPCMStatus TPCM;
414 void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef *CanHandle)
415 {
416     CAN_RxHeaderTypeDef RxHeader;
417     uint8_t i,RxData[8];
418     uint16_t switchValue;
419     CanTxFrameDef FlowControlFrame = {0,0,CAN_ID_STD,CAN_RTR_DATA,8,0x30,0x00,0x00,0x00,0x00,0x00,0x00};
420
421     if (HAL_CAN_GetRxMessage(CanHandle, CAN_RX_FIFO0, &RxHeader, RxData) == HAL_OK)
422     {
423         if(OBDFlag.ProType == SAEJ1939 && RxData[0] == 0x20 && (RxHeader.ExtId&0xffff0000) == 0xEC0000)
424         {
425             CAN_RxRAM[0] = RxData[1];
426             MUXFrame.MaxValue = RxData[3];
427             MUXFrameFlagBool = SET;
428             MUXFrameCount = 1;
429             TPCM = TPCM_BAM;
430         }
431         else if(OBDFlag.ProType == SAEJ1939 && MUXFrameFlagBool == SET && RxData[0] == MUXFrameCount && (RxHeader.ExtId&0xffff0000) == 0xEB00)
432         {
433             for(i = 0; i < 7; i++)
434             {
435                 CAN_RxRAM[i+1+(RxData[0]-1)*7] = RxData[i+1];
436             }
437             if(MUXFrame.MaxValue == MUXFrameCount)
438             {
439                 MUXFrameFlagBool = RESET;
440             }
441         }
442     }
443 }

```

流控帧定义

因为我们的C300车联网开发板采用HAL库编写，在HAL库里面，中断函数都被处理成回调函数的形式，类似于事件。CAN中断的回调函数HAL_CAN_RxFifo0MsgPendingCallback，所以我们的处理代码都在这个函数内进行。如图所示，419行我定义了一个流控帧结构体变量，其实它只对数据域的第一字节进行赋值0x30，标识符并没有赋值，从前面的协议解读中我们知道，流控帧的标识符是根据响应数据的标识符来得到它的物理性请求地址的标识符。421行是HAL库里面的函数，用于接收CAN中断的CAN数据，对于我们有用的数据是最后两个参数，RxHeader存储的是当前接收CAN数据的标识符，IDE，RTR，DLC等信息，RxData存储的是数据域。所以这跟固件函数库有点区别，它把标识符和数据域分开存储在不同的变量里，这些变量需要我们提前定义好。

由于C300开发板编写的HAL_CAN_RxFifo0MsgPendingCallback函数代码较长，不能在一个截图中完整呈现，上图的代码基本是用于SAEJ1939协议的，下面我们部分截取关于ISO15765-4协议的响应数据处理代码进行讲解。

1. SingleFrame 响应数据处理。

```

532 else if((OBDFlag.ProType == ISO15765_4STD_500K && MUXFrameFlagBool == RESET) || (OBDFlag.ProType == ISO15765_4EXT_500K && MUXFrameFlagBool == RESET))
533 {
534     for(i = 0; i < RxHeader.DLC; i++)
535     {
536         CAN_RxRAM[i] = RxData[i];
537     }
538     OBDFlag.RxFlag = NL_SUCCESS;
539 }

```

532行判断全局变量OBDFlag.ProType只要是ISO15765_4STD_500K或者ISO15765_4EXT_500K，并且MUXFrameFlagBool是RESET的话，说明当前接收的是SingleFrame的数据。MUXFrameFlagBool是个全局变量，用在CAN中断内部使用。用来指示当前接收的数据是SingleFrame还是MultipleFrame。MUXFrameFlagBool值是RESET表示SingleFrame，MUXFrameFlagBool值是SET表示MultipleFrame。534行到537行通过RxHeader的DLC信息利用for循环，把RxData接收到的数据存储在CAN_RxRAM数组中。NL_OBD_SendCANFrame函数的返回值是个指针，这个指针指向的地址正是这个CAN_RxRAM数组的地址。存储完毕后，为了告知NL_OBD_SendCANFrame函数退出等待，让OBDFlag.RxFlag赋值NL_SUCCESS。

2. MultipleFrame 响应数据处理

```

488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
}
else if ((OBDFlag.ProType == ISO15765_4STD_500K && RxData[0] == 0x10) || (OBDFlag.ProType == ISO15765_4EXT_500K && RxData[0] == 0x10))
{
    for (i = 0; i < 7; i++)
    {
        CAN_RxRAM[i] = RxData[i+1];
    }
    if((RxData[1]+1)%7 != 0)
    {
        MUXFrameMaxValue = (RxData[1]+1)/7+1;
    }
    else
    {
        MUXFrameMaxValue = (RxData[1]+1)/7;
    }
    MUXFrameFlagBool = SET;
    MUXFrameCount = 0x21;
    FlowControlFrame.IDE = RxHeader.IDE;
    FlowControlFrame.StdId = RxHeader.StdId - 8;
    FlowControlFrame.ExtId = 0x18DA00F1 | ((RxHeader.ExtId & 0xFF)<<8);
    CAN1_Transmit(&FlowControlFrame);
}
else if ((OBDFlag.ProType == ISO15765_4STD_500K && MUXFrameFlagBool == SET && RxData[0] == MUXFrameCount) ||
(OBDFlag.ProType == ISO15765_4EXT_500K && MUXFrameFlagBool == SET && RxData[0] == MUXFrameCount))
{
    for(i = 0; i < 7; i++)
    {
        CAN_RxRAM[7+(RxData[0]-0x20-1)*7+i] = RxData[i+1];
    }
    if(MUXFrameMaxValue == ++MUXFrameCount-0X20)
    {
        MUXFrameFlagBool = RESET;
        OBDFlag.RxFlag = NL_SUCCESS;
    }
}
}

```

488 行判断全局变量 OBDFlag.ProType 只要是 ISO15765_4STD_500K 或者 ISO15765_4EXT_500K, 并且数据域的第 1 字节是 0x10, 则说明当前接收的是 FirstFrame。FirstFrame 有很多 MultipleFrame 特征信息, 所以 490 行到 508 行对 FirstFrame 的处理很重要。490 行到 493 行把 FirstFrame 存储在数组 CAN_RxRAM 里。494 行到 501 行通过 FirstFrame 第 2 个字节计算 ConsecutiveFrame 有多少帧, 计算结果存储在变量 MUXFrameMaxValue 中。502 行设置 MUXFrameFlagBool=SET, 标志要处理 MultipleFrame。503 行设置 ConsecutiveFrame 首字节的比较值, 从 0x21 开始计数。504 行根据当前响应数据设置流控帧的 IDE 类型。505 行和 506 行分别根据响应数据计算流控帧的物理性请求标识符。507 行发送流控帧。接下来的 509 行到 521 行用来接收并存储 ConsecutiveFrame 中的应用层原始数据。509 行和 510 行判断全局变量 OBDFlag.ProType 只要是 ISO15765_4STD_500K 或者 ISO15765_4EXT_500K, 并且 MUXFrameFlagBool 值是 SET, 还有接收的 CAN 数据域第 1 字节 RxData[0] 和 MUXFrameCount 值一致, 就确认为 ConsecutiveFrame。512 行到 515 行提取 ConsecutiveFrame 后 7 个字节进行存储在 CAN_RxRAM。516 行通过判断 MUXFrameMaxValue 和 MUXFrameCount-0x20 计数值是否一致判断接收 ConsecutiveFrame 数据是否完毕。完毕则恢复 MUXFrameFlagBool 值是 RESET, 并且设置 OBDFlag.RxFlag = NL_SUCCESS 说明 MultipleFrame 响应数据处理完毕。

在 NL_OBD_SendCANFrame 函数的支持下, OBD 诊断程序的开发就会变得简单, 我们只需关注 ISO15031-5 协议的功能实现即可, 网络层, 数据链路层等底层处理由 NL_OBD_SendCANFrame 函数完成。下面是 C300 车联网开发板第一版软件关于 OBD 诊断程序代码截图。

```

80 |
81 |     else if(OBDStruct.SYSValue == ISO15765_4STD_500K || OBDStruct.SYSValue == ISO15765_4EXT_500K)
82 |     {
83 |         if(ISO15765_4_WakeUp(CAN_500K,OBDStruct.SYSValue) == NL_OK) ← ISO15765-4初始化函数
84 |         {
85 |             NL_LED_ONOFF(LEDOBD,ON,Fashing,LED1HZ);
86 |             while(1)
87 |             {
88 |                 if(OBDStruct.VINStruct.flag == RESET)
89 |                 {
90 |                     NL_ClearRAM((uint8_t*)OBDStruct.VINStruct.VIN,18);
91 |                     ram = ISO15765_4_GetVIN(OBDStruct.SYSValue,&err);
92 |                     if(err == NL_OK)
93 |                     {
94 |                         strncpy(OBDStruct.VINStruct.VIN,(const char*)ram,17);
95 |                         OBDStruct.VINStruct.flag = SET;
96 |                     }
97 |                 }
98 |                 if(ISO15765_4_GetNotDrivingState(OBDStruct.SYSValue) == NL_OK && OBDStruct.DTCStruct.flag == RESET)
99 |                 {
100 |                     ram = ISO15765_4_GetDTC(OBDStruct.SYSValue,&err); ← 读故障码函数
101 |                     if(err == NL_OK)
102 |                     {
103 |                         NL_ClearRAM((uint8_t*)OBDStruct.DTCStruct.DTC,100);
104 |                         strcpy(OBDStruct.DTCStruct.DTC,ram);
105 |                         OBDStruct.DTCStruct.flag = SET;
106 |                     }
107 |                 }
108 |                 if(OBDStruct.DSStruct.flag == RESET)
109 |                 {
110 |                     dsram = ISO15765_4_GetDS(OBDStruct.SYSValue,ISODSItem,&err); ← 读数据流函数
111 |                     if(err == NL_OK)
112 |                     {
113 |                         OBDStruct.LINKSTATUS = SET;
114 |                         OBDStruct.DSStruct.Total = dsram->Total;
115 |                         for(i = 0; i < OBDStruct.DSStruct.Total; i++)
116 |                         {
117 |                             strncpy(OBDStruct.DSStruct.DS[i],dsram->DS[i],30);
118 |                         }
119 |                         OBDStruct.DSStruct.flag = SET;
120 |                     }
121 |                     else
122 |                     {
123 |                         OBDStruct.LINKSTATUS = RESET;
124 |                         NL_LED_ONOFF(LEDOBD,OFF,Fashing,LED1HZ);
125 |                         break;
126 |                     }
127 |                     RTT_OBD_Printf();
128 |                     LOS_TaskDelay(150);
129 |                 }
130 |             }
131 |         }

```

图中我们看到第一版本软件主要实现读车架号，读故障码，读数据流三大基本 OBD 诊断信息，同时还有一个协议初始化函数。下面我们就围绕这四方面应用程序代码进行具体分析讲解。

ISO15765-4 初始化

图中代码片段 82 行通过调用 ISO15765_4_WakeUp 函数，如果它的返回值是 NL_OK 表示初始化成功。下面截图分析 ISO15765_4_WakeUp 函数源码，了解初始化的过程。

```

15 //*****系统激活*****
16 CanTxFrameDef EntCmd15765 = {0x7DF,0x18DB33FL,CAN_ID_STD,CAN_RTR_DATA,8,0x02,0x01,0x00,0x00,0x00,0x00,0x00,0x00};
17 //*****读故障码*****
18 CanTxFrameDef DTCmd15765 = {0x7DF,0x18DB33FL,CAN_ID_STD,CAN_RTR_DATA,8,0x01,0x03,0x00,0x00,0x00,0x00,0x00,0x00};
19 //*****读数据流*****
20 CanTxFrameDef VinCmd15765 = {0x7DF,0x18DB33FL,CAN_ID_STD,CAN_RTR_DATA,8,0x02,0x09,0x02,0x00,0x00,0x00,0x00,0x00};
21 //*****读车架号*****
22 CanTxFrameDef DSCmd15765 = {0x7DF,0x18DB33FL,CAN_ID_STD,CAN_RTR_DATA,8,0x02,0x01,0x00,0x00,0x00,0x00,0x00,0x00};
23 */
24 * @描述: ISO15765唤醒判断
25 * @参数: NONE
26 * @返回值: NL_OK:支持 NL_NOK:不支持
27 *****/
28 NLStatus ISO15765_4_WakeUp(uint8_t s,PROTypeDef pro)
29 {
30     NLStatus err;
31     if(pro == ISO15765_4STD_500K)
32     {
33         NL_OBD_SelectCAN(CAN_6_14);
34         NL_OBD_CANConfig(s,CAN_ID_STD,0x7e0,0xe0,0x7f0,0x7f0);
35         NL_OBD_SendCANFrame(pro,&EntCmd15765,800,&err);
36     }
37     else
38     {
39         NL_OBD_SelectCAN(CAN_6_14);
40         NL_OBD_CANConfig(s,CAN_ID_EXT,0x18DA0000,0x18DA0000,0x1FFF0000,0x1FFF0000);
41         NL_OBD_SendCANFrame(pro,&EntCmd15765,800,&err);
42     }
43     _NL_Delay(150);
44     return err;
45 }

```

01 00

ISO15765_4_WakeUp 函数有两个参数，参数 1 是 CAN 通信速率，其值我已经用宏定义进行了定义分别是 CAN_500K，CAN_250K，CAN_125K，这三个速率已经涵盖了法规诊断协议和增强型诊断协议也就是专车私有协议的速率。更多速率可以通过计算分频值获得，这部分内容在第 6 章的 6.1.2 小节 CAN 配置中学习。参数 2 是诊断协议类型，只能填 ISO15765_4STD_500K 和 ISO15765_4EXT_500K 这两个协议。

31 行对协议类型进行判断，如果是 ISO15765_4STD_500K 标准 CAN 协议执行 33 行到 35 行。

如果是 ISO15765_4EXT_500K 扩展 CAN 协议执行 39 行到 41 行。首先看标准 CAN 协议 33 行 NL_OBD_SelectCAN 函数，这个函数是用来选择 OBD 接口 CAN 通信引脚的，参数分别是有 CAN_6_14 和 CAN_3_11。CAN_6_14 选择 OBD 接口的 6 和 14 引脚作为 CAN 通信，6 是 CANH, 14 是 CANL。CAN_3_11 选择 OBD 接口的 3 和 11 引脚作为 CAN 通信，3 是 CANH, 11 是 CANL。对于法规协议 ISO15765-4 协议，CAN 通信引脚被固定定义为 OBD 接口的 6 和 14 引脚，所以此处函数 NL_OBD_SelectCAN (CAN_6_14)。34 行 NL_OBD_CANConfig 函数主要用于设置 CAN 通信的速率，CAN 的 IDE 类型和滤波器，第 1 个参数用于设置 CAN 速率，第 2 参数用于设置 CAN 的 IDE 类型，也就是标准 CAN 通信还是扩展 CAN 通信，第 3 个参数用于设置滤波器，当前滤波器设置值为 0x7e0, 0x7e0, 0x7f0, 0x7f0，这个值的设置和我们的 OBD 分析仪上位机滤波器设置是一摸一样的，由两组滤波器组成，四个值分别代表 滤波器 1 屏蔽码，滤波器 2 屏蔽码，滤波器 1 校验码，滤波器 2 校验码，因为在 ISO15765-4 协议关于标识符定义我们知道，标准 CAN 效应数据的高 7 位值都固定是 0x7e，所以过滤器屏蔽码 0x7e0，而效验码 0x7f0，只比较高 7 位是否是 0x7e，低 4 位忽略即可。35 行通过 NL_OBD_SendCANFrame 函数发送 EntCmd15765 请求数据，注意这个请求数据的 SID=0x01, PID=0x00, 这是 ISO15765-4 协议推荐我们使用的初始化请求数据，也是 ISO15031-5 协议定义的 Supported PID 请求数据，而当前版本软件我们并没有做 Supported PID 列表，这将在第二版软件根据实际需求添加上该部分代码。如果 NL_OBD_SendCANFrame 函数第 3 个参数值 err=NL_OK, 说明接收到了针对 SID=0x01, PID=0x00 的响应数据，证明当前汽车支持 ISO15765_4STD_500K 协议。接下来的 ISO15765_4EXT_500K 扩展 CAN 协议基本类似，39 行和 41 行函数和标准 CAN 类似。这里主要讲解下 40 行的扩展 CAN 滤波器设置，根据 ISO15765-4 关于扩展 CAN 请求和响应标识符的定义，响应数据的标识符高 13 位是不变的，低 16 位是两个字节，分别表示目标地址和源地址，传输方向的不同，且物理硬件的不同，值都会发生变化。所以只有高 13 位 0x18DA 是固定的。针对这个情况滤波器设置值为 0x18DA0000, 0x18DA0000, 0x1FFF0000, 0x1FFF0000，两组滤波器过滤判断一样，比较高 13 位的值必须是 0x18DA，低 16 位忽略比较。

读车架号

下面截图是读车架号的代码片段。

```

87 if(OBDStruct.VINStruct.flag == RESET)
88 {
89     NL_ClearRAM((uint8_t*)OBDStruct.VINStruct.VIN, 18);
90     ram = ISO15765_4_GetVIN(OBDStruct.SYSValue, &err);
91     if(err == NL_OK)
92     {
93         strncpy(OBDStruct.VINStruct.VIN, (const char*)ram, 17);
94         OBDStruct.VINStruct.flag = SET;
95     }
96 }
```

87 行判断 OBDStruct.VINStruct.VIN 是否存储了读到的车技号，如果 OBDStruct.VINStruct.flag 的值是 RESET 表示 OBDStruct.VINStruct.VIN 值是空的或者没有更新的；OBDStruct.VINStruct.flag 的值是 SET 表示读到了车架号并存储在 OBDStruct.VINStruct.VIN 中。同时 OBDStruct.VINStruct.flag 也是同步 TaskTCP 的变量，OBDStruct.VINStruct.flag 值是 SET 时，TaskTCP 才会上传车架号到服务器。89 行清空 OBDStruct.VINStruct.VIN 变量。90 行读取车架号，并返回存储车架号地址到 ram 指针。91 行到 94 行车架号读取成功把 ram 指向的存储单元存储的车架号复制到 OBDStruct.VINStruct.VIN 中，并设置 OBDStruct.VINStruct.flag = SET 告知 TaskTCP 可以上传车架号信息到服务器。

如何请求和响应车架号，我们要看 ISO15765_4_GetVIN 函数。

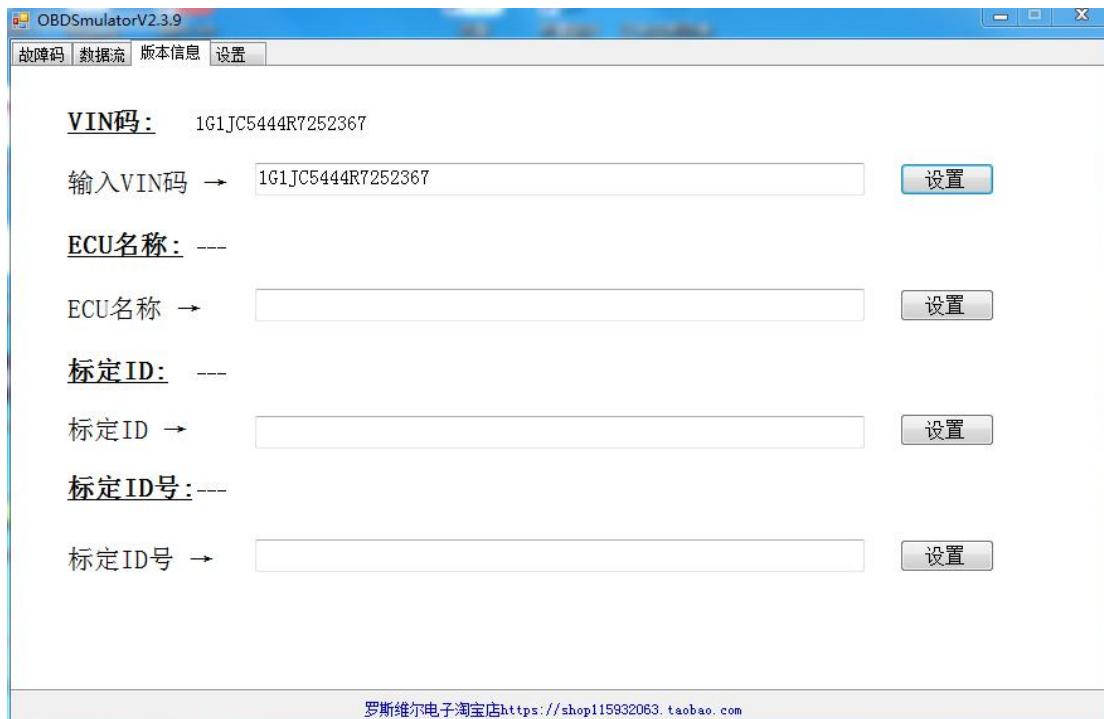
```

46 /**
47  * @描述: ISO15765获取车架号
48  * @参数: pro: 协议类型 ISO15765_4STD_500K or ISO15765_4EXT_500K,*err :NL_OK:成功 NL_NOK:不成功
49  * @返回值: 车架号存储指针
50 */
51 char VINRAM[18];
52 char* ISO15765_4_GetVIN(ProtocolDef pro,NLStatus *err)
53 {
54     uint8_t i,*ram;
55     ram = NL_OBD_SendCANFrame(pro,&VinCmd15765,800,err);
56     _NL_Delay(150);
57     if(*err == NL_OK)
58     {
59         for(i = 0; i < 17; i++)
60         {
61             VINRAM[i] = ram[i+4];
62         }
63     }
64     VINRAM[17] = 0;
65     return VINRAM;
66 }

```

函数有两个参数，参数 1 表示协议类型，参数 2 *err=NL_OK 表示函数读取车架号成功，*err=NL_NOK 表示读取车架号失败。55 行利用 NL_OBD_SendCANFrame 函数发送车架号请求，并获得汽车或者模拟器的响应数据。下面进行模拟并在线仿真帮助理解。

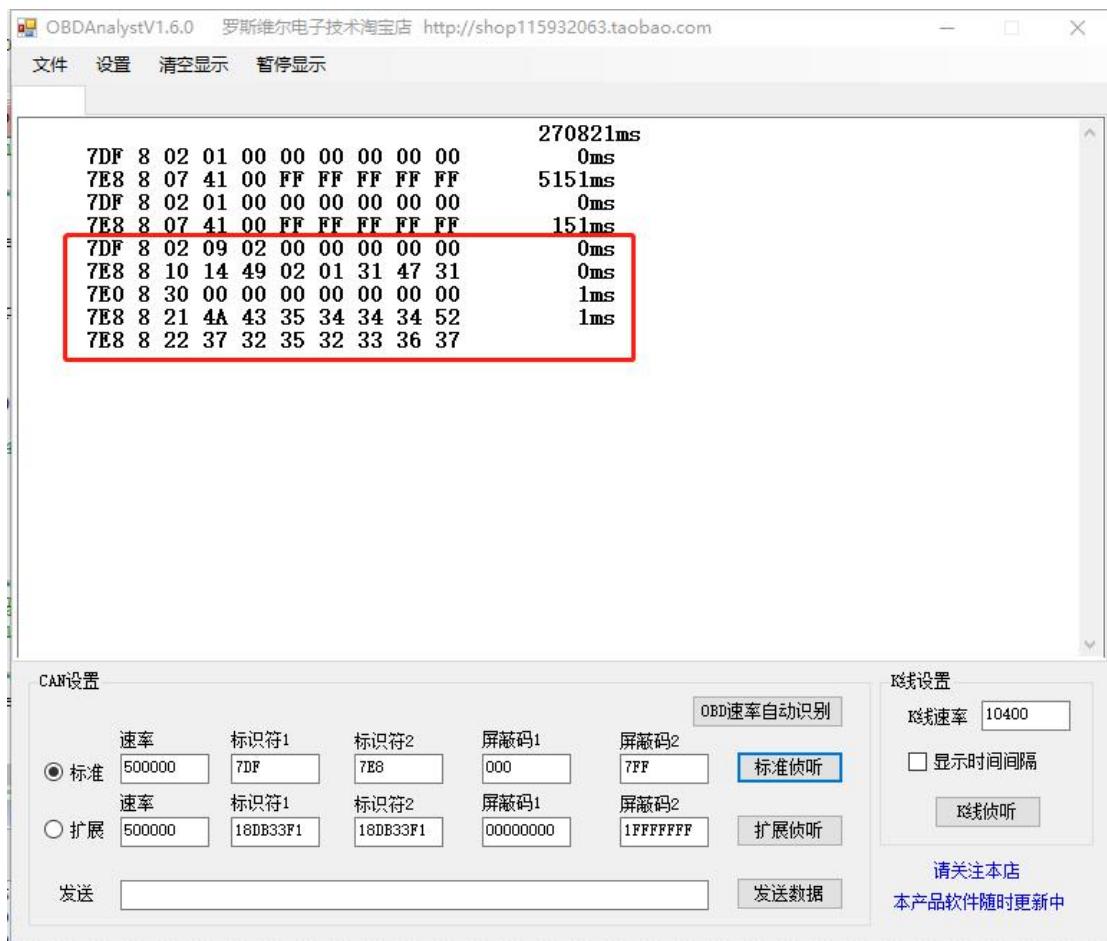
1. 设置模拟器模拟车架号 1G1JC5444R7252367



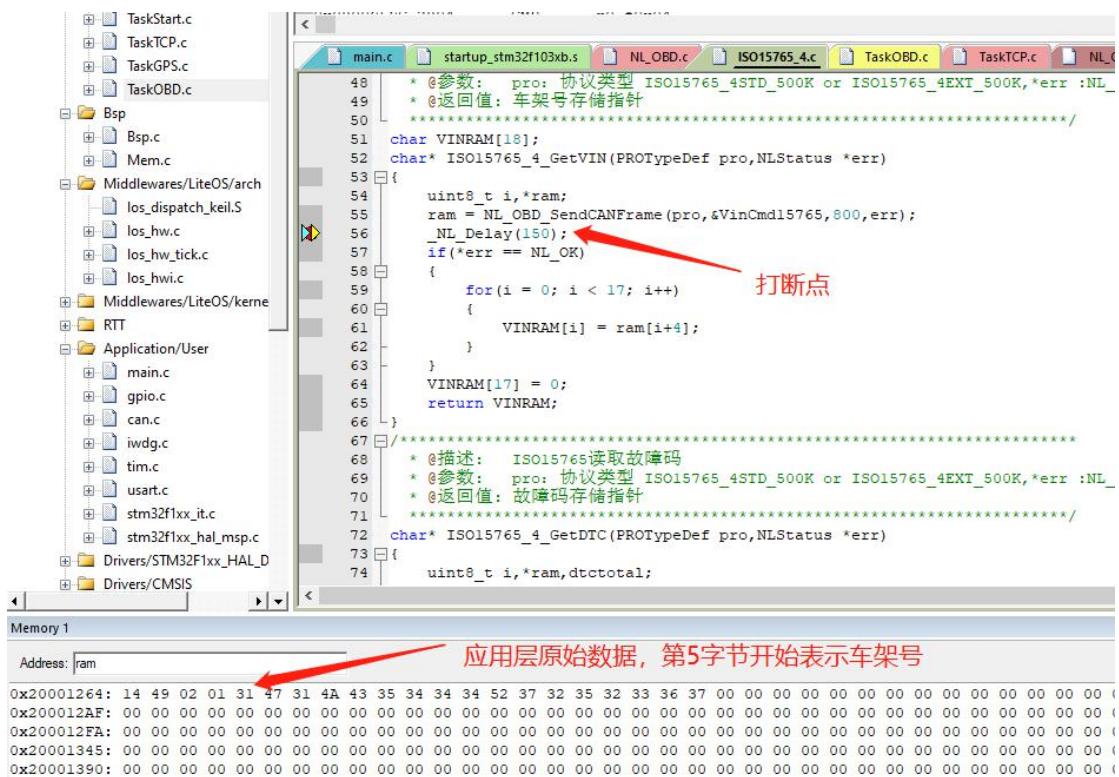
2. C300 车联网开发板对应的服务器网站显示读到的正确车架号。



3. OBD 分析仪采集到的读车架号原始数据。



4. 打断点，查看 ram 指针对应存储单元存储的响应数据。应用层原始数据从第 5 字节开始表示车架号。



5. 下图把 Memory 区转换成 ASCII 编码，可以清晰查看到车架号。

The screenshot shows a software development environment with a file tree on the left and a code editor and memory dump window on the right.

File Tree:

- TaskStart.c
- TaskTCP.c
- TaskGPS.c
- TaskOBD.c
- Bsp
 - Bsp.c
 - Mem.c
- Middlewares/LiteOS/arch
 - los_dispatch_keil.S
 - los_hw.c
 - los_hw_tick.c
 - los_hwic.c
- Middlewares/LiteOS/kerne
- RTT
- Application/User
 - main.c
 - gpio.c
 - can.c
 - iwdg.c
 - tim.c
 - usart.c
 - stm32f1xx_it.c
 - stm32f1xx_hal_msp.c
- Drivers/STM32F1xx_HAL_D
- Drivers/CMSIS

Code Editor (main.c):

```

48 * @参数: pro: 协议类型 ISO15765_4STD_500K or ISO15765_4EXT_500K,*err :NL_OK:成功 NL_NOI
49 * @返回值: 车架号存储指针
50 ****
51 char VINRAM[18];
52 char* ISO15765_4_GetVIN(PROTypeDef pro,NLStatus *err)
53 {
54     uint8_t i,*ram;
55     ram = NL_OBD_SendCANFrame(pro,&VinCmd15765,800,err);
56     _NL_Delay(150);
57     if(*err == NL_OK)
58     {
59         for(i = 0; i < 17; i++)
60         {
61             VINRAM[i] = ram[i+4];
62         }
63     }
64     VINRAM[17] = 0;
65     return VINRAM;
66 }
67 ****
68 * @描述: ISO15765读取故障码
69 * @参数: pro: 协议类型 ISO15765_4STD_500K or ISO15765_4EXT_500K,*err :NL_OK:成功 NL_NOI
70 * @返回值: 故障码存储指针
71 ****
72 char* ISO15765_4_GetDTC(PROTypeDef pro,NLStatus *err)
73 {
74     uint8_t i,*ram,dttctotal;

```

Memory Dump (Memory 1):

Address	Value
0x20001264	.I..1G1JC5444R7252367
0x20001345	03,,,"36.....
0x20001426	03,,,"36.....
0x20001507
0x200015E8

A red arrow points from the value at address 0x20001345 to the text "切换成ASCII编码可以清楚看到车架号" (Switch to ASCII encoding to clearly see the chassis number).

所以 57 行到 63 行代码就是把 ram 指向的存储单元第 5 个字节开始 17 个字节表示车架号复制到 VINRAM 数组中。64 行代码是在 VINRAM 数组车架号后加入 0 这个结束字符。65 行代码返回 VINRAM 数组地址。这就是 ISO15765-4 协议车架号读取过程。

读故障码

读故障码代码片段如下所示。

```

97
98 if(ISO15765_4_GetNotDrivingState(OBDStruct.SYSValue) == NL_OK && OBDStruct.DTCStruct.flag == RESET)
99 {
100     ram = ISO15765_4_GetDTC(OBDStruct.SYSValue,&err);
101     if(err == NL_OK)
102     {
103         NL_ClearRAM((uint8_t*)OBDStruct.DTCStruct.DTC,100);
104         strcpy(OBDStruct.DTCStruct.DTC,ram);
105         OBDStruct.DTCStruct.flag = SET;
106     }
}

```

97 行判断车速是否为 0，通过函数 ISO15765_4_GetNotDrivingState 判断。并且判断 OBDStruct.DTCStruct.flag 是否为 RESET。条件满足的情况下才能读取故障码。其中 OBDStruct.DTCStruct.flag 值是 RESET 说明刚上电从来没有读取故障码，或者 OBDStruct.DTCStruct.DTC 的故障码已经被 TaskTCP 上传至服务器。99 行是读取故障码函数，接下来我们会具体分析这个函数。100 行到 105 行判断 ISO15765_4_GetDTC 函数是否读取故障码成功，如果读取成功就把 ISO15765_4_GetDTC 函数返回的指针指向存储单元的故障码存储到 OBDStruct.DTCStruct.DTC 中，并且设置 OBDStruct.DTCStruct.flag = SET 供 TaskTCP 上传至服务器。

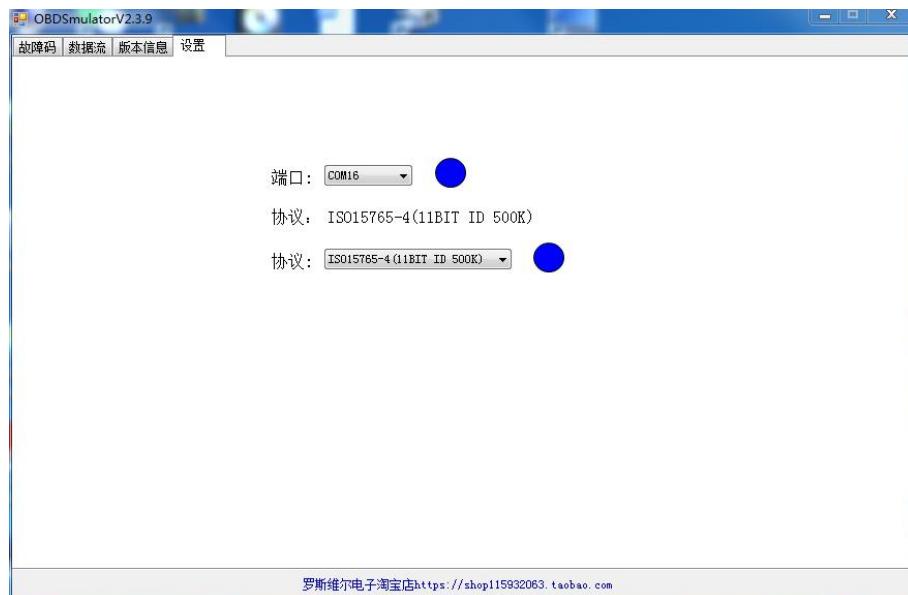
下面具体看 ISO15765_4_GetDTC 函数。

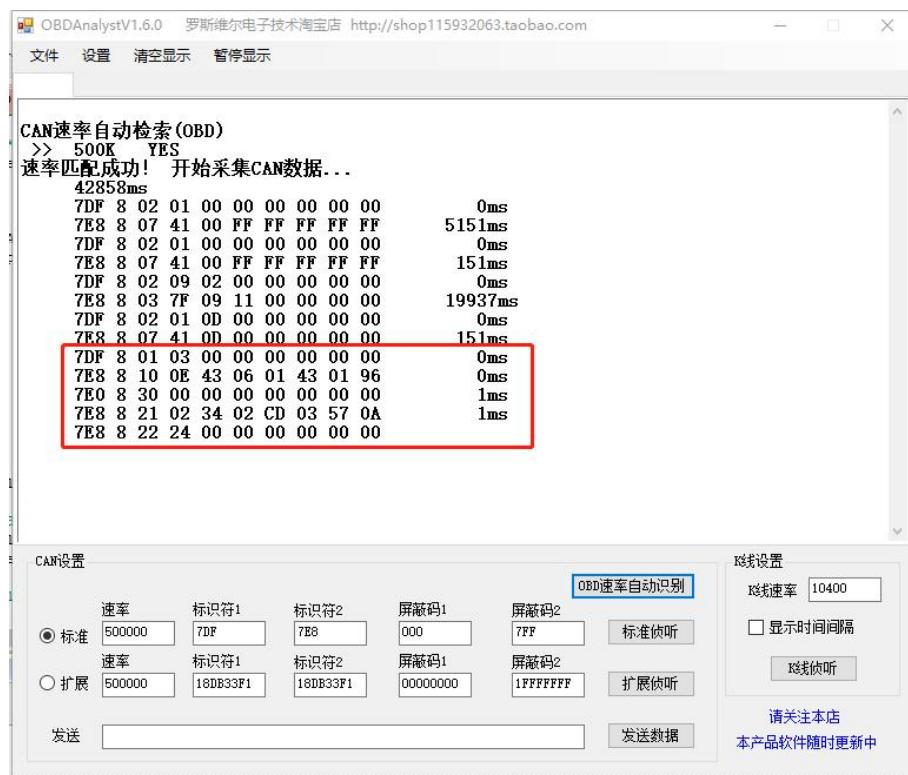
```

67  /*************************************************************************/
68  * @描述: ISO15765读取故障码
69  * @参数: pro: 协议类型 ISO15765_4STD_500K or ISO15765_4EXT_500K,*err :NL_OK:成功 NL_NOK:不成功
70  * @返回值: 故障码存储指针
71  ****
72  char* ISO15765_4_GetDTC(ProtocolDef pro,NLStatus *err)
73  {
74      uint8_t i,*ram,dtctotal;
75      uint16_t dtc;
76      NL_ClearRAM((uint8_t*)DTCRAM,200);
77      ram = NL_OBD_SendCANFrame(pro,&DTCCmd15765,800,err);
78      NL_Delay(150);
79      if(*err == NL_OK)
80      {
81          if(ram[2] >10)
82          {
83              dtctotal = 10;
84          }
85          else
86          {
87              dtctotal = ram[2];
88          }
89          for(i = 0; i < dtctotal; i++)
90          {
91              dtc = ram[3+i*2]<<8|ram[4+i*2];
92              strcpy(DTCRAM+strlen(DTCRAM),(const char*)NL_PCBU(dtc),5);
93              strcpy(DTCRAM+strlen(DTCRAM),";");
94          }
95          DTCRAM[strlen(DTCRAM)-1] = 0;
96      }
97      return DTCRAM;
98  }

```

函数有两个参数，参数 1 表示协议类型。参数 2 表示故障码读取成功与否的标志。76 行代码对 DTCRAM 进行清空，DTCRAM 在本函数中用于存储故障码结果。77 行利用 NL_OBD_SendCANFrame 函数发送故障码请求并希望获得汽车或者模拟器响应，返回指针指向故障码响应数据的应用层原始数据的存储单元。为了更直观理解函数工作原理，使用模拟器设置故障码 P0143, P0196, P0234, P02CD, P0357, P0A24，并在 78 行打断点查看 ram 指向存储单元的值，如下图所示。





```

main.c startup_stm32f103xs.c NL_OBD.c ISO15765_4.c TaskOBD.c TaskTCP.c NL_OBD.h gfp

67 /* @描述: ISO15765读取故障码
68 * @参数: pro: 协议类型 ISO15765_4STD_500K or ISO15765_4EXT_500K, *err :NL_OK:成功 NI
69 * @返回值: 故障码存储指针
70 */
71 ****
72 char* ISO15765_4_GetDTC(ProtocolDef pro, NLStatus *err)
73 {
74     uint8_t i,*ram,dtctotal;
75     uint16_t dtc;
76     NL_ClearRAM((uint8_t*)DTCRAM,200);
77     ram = NL_OBD_SendCANFrame(pro,&DTCCmd15765,800,err);
78     _NL_Delay(150);
79     if(*err == NL_OK)
80     {
81         if(ram[2] >10)
82         {
83             dtctotal = 10;
84         }
85         else
86         {
87             dtctotal = ram[2];
88         }
89         for(i = 0; i < dtctotal; i++)
90         {
91             dtc = ram[3+i*2]<<8|ram[4+i*2];
92             strcpy(DTCRAM+strlen(DTCRAM),(const char*)NL_PCBU(dtc),5);
93             strcpy(DTCRAM+strlen(DTCRAM),"");
94         }
95         DTCRAM[strlen(DTCRAM)-1] = 0;
96     }
97     return DTCRAM;
98 }
99 ****
100 /* @描述: ISO15765读取数据流

```

Memory 1

Address: ram

响应数据的应用层原始数据

```

0x200012640: 0E 43 06 01 43 01 96 02 34 02 CD 03 57 0A 24 00
0x20001264F: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x200012650: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x200013450: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x20001345F: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

```

ram 指向存储单元这些值中，第 3 字节表示故障码个数，第 4 字节开始每两个字节表示一个故障码。所以下面就很容易理解了，81 行到 88 行限制故障码个数，最多故障码个数不能超过 10 个，ram[2] 存储的就是第 3 字节故障码个数。89 到 94 行以故障码个数为 for 循环次数，循环一次把一个应用层原始数据的故障码编码成 ascii 编码的故障码。结果存储于 DTCRAM 中，我们可以在 97 行打断点看看处理的结果，如下图所示。

```

main.c startup_stm32f103xs.c NL_OBD.c ISO15765_4.c TaskOBD.c TaskTCP.c NL_OBD.h gfp

65 }
66 ****
67 /* @描述: ISO15765读取故障码
68 * @参数: pro: 协议类型 ISO15765_4STD_500K or ISO15765_4EXT_500K, *err :NL_OK:成功 NI
69 * @返回值: 故障码存储指针
70 */
71 ****
72 char* ISO15765_4_GetDTC(ProtocolDef pro, NLStatus *err)
73 {
74     uint8_t i,*ram,dtctotal;
75     uint16_t dtc;
76     NL_ClearRAM((uint8_t*)DTCRAM,200);
77     ram = NL_OBD_SendCANFrame(pro,&DTCCmd15765,800,err);
78     _NL_Delay(150);
79     if(*err == NL_OK)
80     {
81         if(ram[2] >10)
82         {
83             dtctotal = 10;
84         }
85         else
86         {
87             dtctotal = ram[2];
88         }
89         for(i = 0; i < dtctotal; i++)
90         {
91             dtc = ram[3+i*2]<<8|ram[4+i*2];
92             strcpy(DTCRAM+strlen(DTCRAM),(const char*)NL_PCBU(dtc),5);
93             strcpy(DTCRAM+strlen(DTCRAM),"");
94         }
95         DTCRAM[strlen(DTCRAM)-1] = 0;
96     }
97     return DTCRAM;
98 }
99 ****
100 /* @描述: ISO15765读取数据流

```

Memory 1

Address: DTCRAM

DTCRAM存储本函数故障码结果

```

0x200014D90: F0143;F0196;F0234;F02cd;F0357;F0a24,.
0x200015B00: .
0x200016B01: .
0x2000177C1: .

```

95 行在 DTCRAM 故障码编码处理完毕后在最后一个字节末尾加上 0 结束字符。97 行函数返回 DTCRAM 数组地址。

读数据流

读数据流代码片段如下所示。

```

107 if(OBDStruct.DSStruct.flag == RESET)
108 {
109     dsram = ISO15765_4_GetDS(OBDStruct.SYSValue,ISODSItem,&err);
110     if(err == NL_OK)
111     {
112         OBDStruct.LINKSTATUS = SET;
113         OBDStruct.DSStruct.Total = dsram->Total;
114         for(i = 0; i < OBDStruct.DSStruct.Total; i++)
115         {
116             strncpy(OBDStruct.DSStruct.DS[i],dsram->DS[i],30);
117         }
118         OBDStruct.DSStruct.flag = SET;
119     }
120     else
121     {
122         OBDStruct.LINKSTATUS = RESET;
123         NL_LED_ONOFF(LED0BD,OFF,Fashing,LED1HZ);
124         break;
125     }
126 }

```

107 行判断是否需要读取数据流，OBDStruct.DSStruct.flag 值是 RESET 时，表示刚上电或者数据流已经被 TaskTCP 上传至服务器，需要重新读取数据流。OBDStruct.DSStruct.flag 值是 SET 时，表示数据流读取完成并存储在 OBDStruct.DSStruct.DS 中，暂时没有被上传至服务器，不需要重新读取数据流。109 行读数据流函数，读取结果存储在 dsram 所指向的存储单元。110 行到 119 行表示成功读取数据流，把读到的数据流结果存储在 OBDStruct.DSStruct.DS 中，并设置全局变量 OBDStruct.DSStruct.flag 和 OBDStruct.LINKSTATUS 为 SET，告知 TaskTCP 上传数据流到服务器。OBDStruct.LINKSTATUS 值是 SET 时，TaskTCP 会保持上传数据流信息到服务器，如果是 RESET 时，暂停上传数据流信息到服务器，只上传 IMEI 信息。所以 120 行到 125 行表示读取数据流失败，OBDStruct.LINKSTATUS 被赋值 RESET，暂停上传数据流信息，并关闭硬件指示 OBD 状态的 LED 灯（123 行），通过 break 语句退出当前循环，这时候被认为汽车已经关闭钥匙（OFF 状态）。

下面具体看 ISO15765_4_GetDS 函数是如何实现读数据流的。

```

99 /**
100  * @描述: ISO15765读取数据流
101  * @参数: pxo 协议类型 ISO15765_4STD_500K or ISO15765_4EXT_500K,wint8_t *item: 数据流项目索引 ,*err :NL_OK:成功 NL_NOK:不成功
102  * @返回值: 数据流存储指针
103 */
104 DSStructDef DSStruct;
105 uint8_t ErrCount = 0;
106 DSStructDef* ISO15765_4_GetDS(PTROTypeDef pro,DSItemStructDef item,NLStatus *err)
107 {
108     uint8_t i,*ram;
109     DSStruct.Total = item.Total;
110     for(i = 0; i < item.Total; i++)
111     {
112         DS Cmd15765.Data[2] = DSControl15301[item.Item[i]].PIDByte;
113         ram = NL_OBD_SendCANFrame(pxo,&DS Cmd15765,800,err);
114         NL_Delay(150);
115         if(*err == NL_OK)
116         {
117             ErrCount = 0;
118             if (DSControl15301[item.Item[i]].Type == Numeric)
119             {
120                 sprintf(DSStruct.DS[i],DSControl15301[item.Item[i]].Format,DSControl15301[item.Item[i]].Equation1(ram+DSControl15301[item.Item[i]].FineByte));
121             }
122             else
123             {
124                 strcpy(DSStruct.DS[i],"");
125                 strcpy(DSStruct.DS[i]+strlen(DSStruct.DS[i]),DSControl15301[item.Item[i]].Equation1(ram+DSControl15301[item.Item[i]].FineByte));
126                 strcpy(DSStruct.DS[i]+strlen(DSStruct.DS[i]),"");
127             }
128         }
129         else
130         {
131             if (++ErrCount > 6)
132             {
133                 *err = NL_NOK;
134                 return &DSStruct;
135             }
136         }
137     }
138     *err = NL_OK;
139 }
140 }
141 }
142 }
143 }
144 return &DSStruct;
145 }

```

ISO15765_4_GetDS 函数有三个参数，参数 1 表示协议类型，参数 2 表示数据流项目索引，C300 车联网开发板需要读什么数据流由这个索引值进行指定，参数 3 读取数据流成功与否的标志。

进入函数体代码解读前，要先了解参数 2 数据流索引才能很好理解这个函数。下图是数据流索引的定义，这仅适用于第一本版软件的定义

```

/*@定义乘用车数据流项目
 ****
 DSItemStructDef ISODSItem = { 6, 48, 49, 40, 41, 47, 51 };
 ****
 /******结构定义*****/
typedef struct
{
    uint8_t Total;
    uint8_t Item[10];
}DSItemStructDef;

```

第 1 个数值 6 是 Total 成员，表示有 6 个数据流。剩下的 6 个值均属于 Item 成员。这 6 个 Item 对应结构体数组 DSControl15301[DSTotalX] 如下图所示。

```

/******数据流控制*****/
const DSControl15301TypeDef DSControl15301[DSTotalX] = {
    {Numeric , 0x01, 3, "%0f", Formula000, NONE }, //DS000 ECU中存储的故障码数量
    {Character, 0x01, 3, "      ", NONE , Formula001}, //DS001 MIL(故障指示灯)状态
    {Character, 0x01, 4, "      ", NONE , Formula002}, //DS002 支持失火监测
    {Character, 0x01, 4, "      ", NONE , Formula003}, //DS003 支持燃油系统监测
    {Character, 0x01, 4, "      ", NONE , Formula004}, //DS004 支持综合部件监测
}

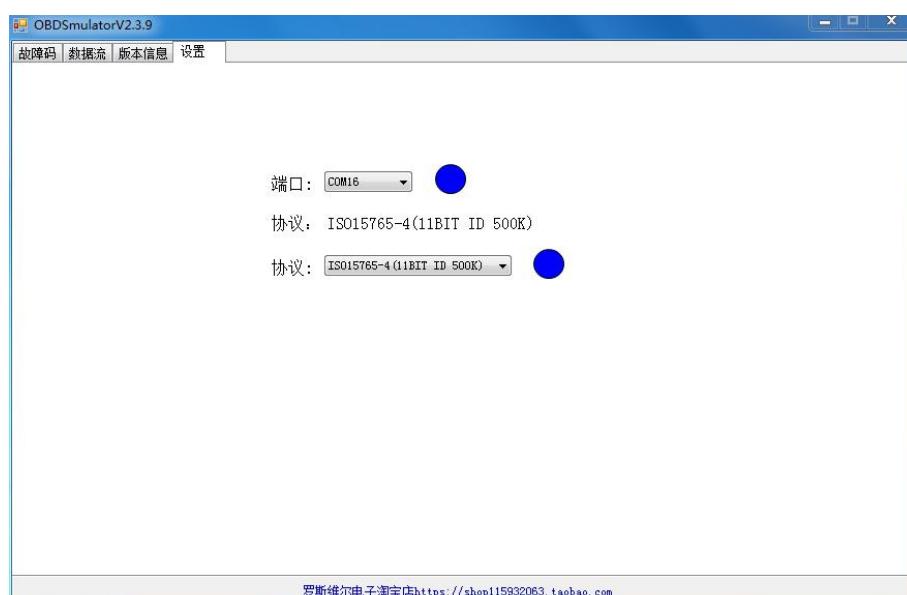
```

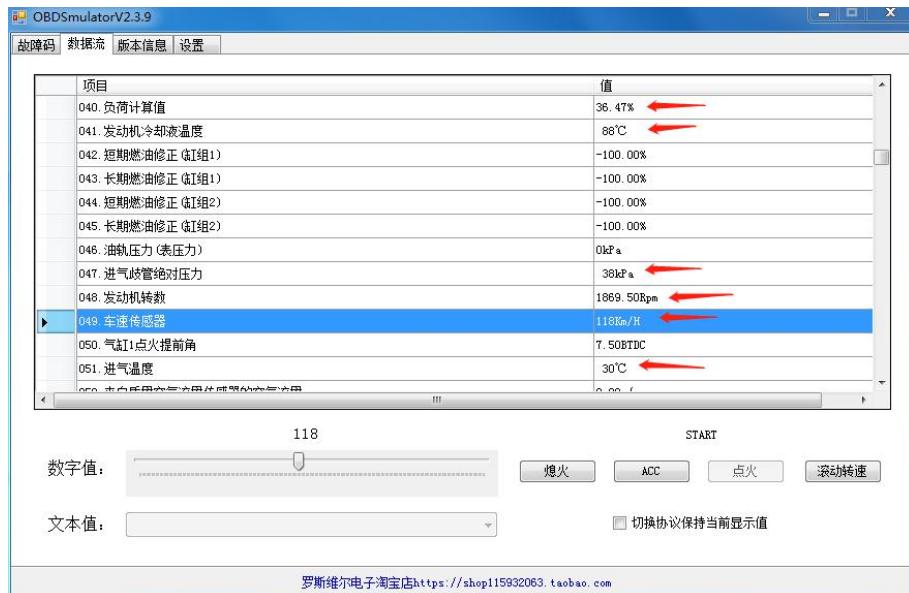
这个数组较长，下面分别截图这 6 个 Item 值 48, 49, 40, 41, 47, 51 所在位置。

{Numeric , 0x04, 3, "%2f", Formula018, NONE }	//DS040 负荷计算值
{Numeric , 0x05, 3, "%0f", Formula019, NONE }	//DS041 发动机冷却液温度
{Numeric , 0x06, 3, "%2f", Formula020, NONE }	//DS042 短期燃油修正(缸组1)
{Numeric , 0x07, 3, "%2f", Formula020, NONE }	//DS043 长期燃油修正(缸组1)
{Numeric , 0x08, 3, "%2f", Formula020, NONE }	//DS044 短期燃油修正(缸组2)
{Numeric , 0x09, 3, "%2f", Formula020, NONE }	//DS045 长期燃油修正(缸组2)
{Numeric , 0x0A, 3, "%0f", Formula021, NONE }	//DS046 油轨压力(表压力)
{Numeric , 0x0B, 3, "%0f", Formula022, NONE }	//DS047 进气歧管绝对压力
{Numeric , 0x0C, 3, "%0f", Formula023, NONE }	//DS048 发动机转数
{Numeric , 0x0D, 3, "%0f", Formula022, NONE }	//DS049 车速传感器
{Numeric , 0x0E, 3, "%2f", Formula024, NONE }	//DS050 气缸1点火提前角
{Numeric , 0x0F, 3, "%0f", Formula019, NONE }	//DS051 进气温度
{Numeric , 0x10, 3, "%2f", Formula025, NONE }	//DS052 来自质量空气流量传感器的空气质量
{Numeric , 0x11, 3, "%2f", Formula018, NONE }	//DS053 节气门绝对位置

理解清楚这个数据流索引之后，我们正式看 ISO15765_4_GetDS 函数代码。109 行把传入参数的 ISODSItem.Total 值赋值给结构体成员 DSStruct.Total，因为 DSStruct 结构体地址会作为本函数的返回值进行返回。110 行到 142 行就是按照 6 个数据流索引进行读取数据流。这里可以利用模拟器打断点看看 DSStruct 结果是否能读到正确的数据流。实验如下图所示。

1. 设置模拟器参数

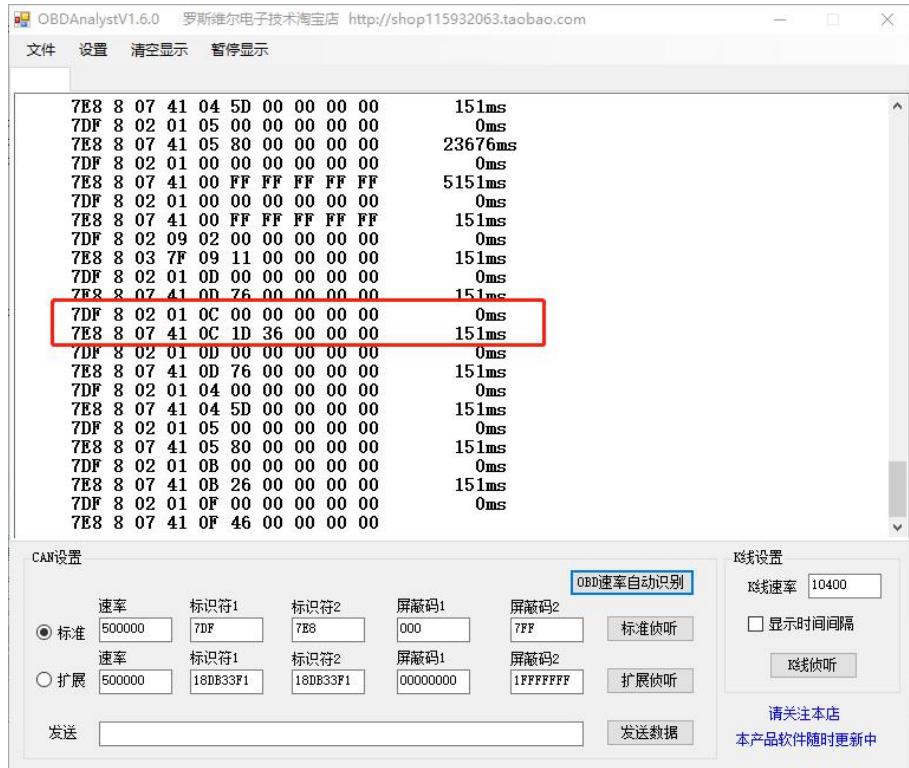




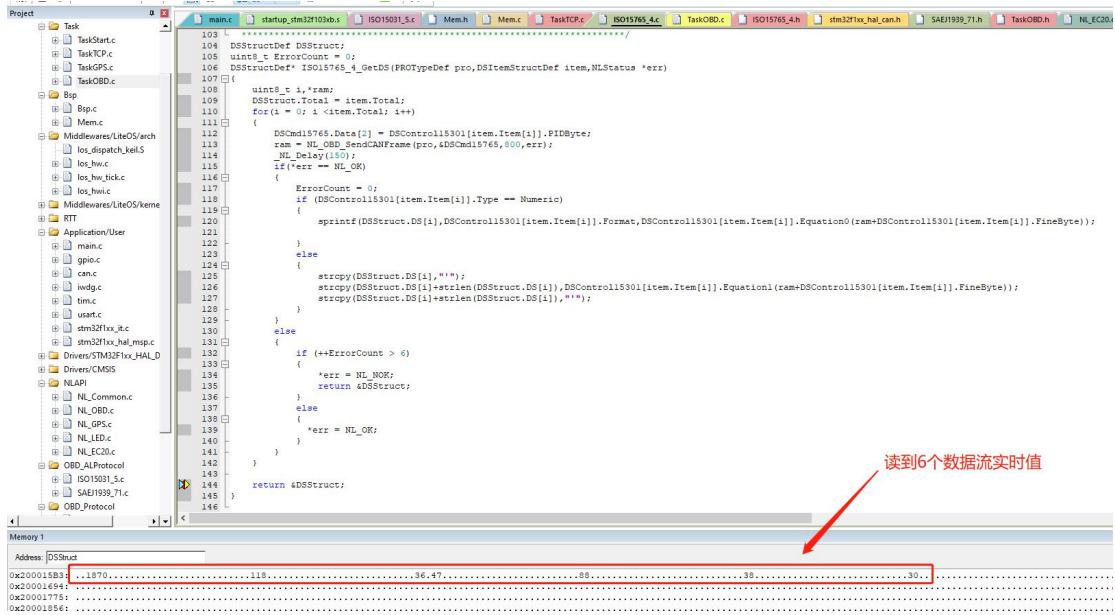
2. C300 车联网开发板采集到的诊断数据



3. OBD 分析仪采集到的 C300 和模拟器间通信的数据



4. 在函数的 144 行打断点，如下图所示。函数可以读到正确的数据流。



112 行从结构体数组 DSControl15301 取出 PID，放入 DSCmd15765 请求命令的 PID 位置，实现了命令 PID 自动填充，以下图发动机转速为例，PID 就是 0x0C。

{Numeric ,0x0C,3,"%Of",Formula023,NONE } //DS048 发动机转数

113 行通过 NL_OBD_SendCANFrame 函数请求并获得响应。115 行到 129 行是成功获得响应之后，对数据流应用层原始数据的处理，把处理结果以 ASCII 形式存储在 DSStruct 中。这里我们只讨论 120 行代码，因为 120 行处理的是数值型数据，而 125 行到 127 行处理的是字符串型数据，这部分数据只上传到数据库，但是我们的网站不显示，只显示 120 行处理的数值型数据。仍然以发动机转速为例解释 120 行这个代码，sprintf 是 C 库函数应用比较广泛，使用它很方便把数值按照一定的格式控制转换成 ASCII 编码，sprintf 函数有三个参数，参数 1 代码 DSStruct.DS[i]，这是把转换的结果以 ASCII 编码形式存储在 DSStruct.DS[i] 中；参数 2 代码 DSControl15301[item.Item[i]].Format，这是格式控制字符，以发动机转速为例就是 "%.Of"；参数 3 代码

DSControl15301[item.Item[i]].Equation0(ram+DSControl15301[item.Item[i]].FineByte)，有点长，分解之后就很容易理解，两部分组成，函数指针和函数参数，首先看函数指针 DSControl15301[item.Item[i]].Equation0()，这是调用数据流转换公式，以发动机转速为例它的值就是 Formula023，函数源码如下图所示。

```
/*
 * @描述: 发动机转数
 * @参数: u8 *data
 * @返回值: float
 */
float Formula023(u8 *p)
{
    return (u16)(*p << 8 | *(p+1)) / 4.0;
}
```

参数*p 就是对应代码 ram+DSControl15301[item.Item[i]].FineByte 以发动机转速为例就是 ram+3 的地址传入参数，而 Formula023 函数会取 ram+3 和 ram+4 两个字节分别表示高字节和低字节组成一个 16 位 2 个字节的数除以 4 获得发动机转速。这里可以对照 ISO15031-5 关于发动机转速的定义，如下图所示完全一致。

Table B.13 — PID \$0C definition

PID (hex)	Description	Data byte	Min. value	Max. Value	Scaling/bit	External test equipment SI (Metric) / English display
0C	Engine RPM	A, B	0 min ⁻¹	16383,75 min ⁻¹	1/4 rpm per bit	RPM: xxxx min ⁻¹
Engine RPM shall display revolutions per minute of the engine crankshaft.						

接下来的 132 行到 140 行是一个冲裁的代码片段, 如果函数请求的 6 个数据流有 5 个没能获得响应返回 NL_NOK, 表示汽车熄火关闭钥匙或者连接断开或者不支持数据流的情况。

7.2.3 ISO15765-4 协议以及在开发板中的程序实现视频教程

7.2.1 ISO15765-4 协议解读视频教程

视频教程链接 <https://www.bilibili.com/video/BV1iE411H7dK?p=12>

7.2.2 ISO15765-4 协议程序实现视频教程

视频教程链接 <https://www.bilibili.com/video/BV1iE411H7dK?p=13>

7.3 ISO14230-4 协议以及在开发板中的程序实现

7.3.1 ISO14230-4 协议解读

ISO14230-4 协议物理层是基于 K 线通信，简单描述 K 线就是把串口收发合并在一根通信线上，同时用 12V 或者 24V 正电表示逻辑 1，0V 表示逻辑 0。对于物理层的具体定义可参考 ISO14230-1 协议，上面严格定义了硬件电路 K 线该如何设计，K 线设计并不只有一个固定的方案，只要满足 ISO14230-1 协议的设计需求即可。Neulen Tbox C300 车联网开发板硬件电路提供了一种分立元件的方案，已经过长期测试，所以这里不再对 ISO14230-1 协议做具体解读。本节 ISO14230-4 程序源码的实现均基于 Neulen Tbox C300 车联网开发板 K 线设计方案。我们围绕一贯的协议解读办法从协议中找到通信如何唤醒，数据格式，通信信息管理，应用层如何定义这四个方面进行解读。

1. 协议通信如何唤醒？

首先要阅读并参考 ISO14230-4 协议的指引进行解读。下图是 ISO14230-4 协议 4.4 小节关于开始通信服务即协议通信初始化的描述。

4.4 StartCommunication service

ECU(s) (OBD related) shall only support one of the two following methods of initialisation:

- 5 baud initialisation;
- fast initialisation.

The scan tool shall support both methods:

- 5 baud initialisation;
- fast initialisation.

Keywords received by the scan tool can be 2025, 2027, 2029 and 2031. In any case, the scan tool and the vehicle shall only use the functionality of keyword 2025 (i.e. 3 byte header, no additional length byte, normal timing).

In the case where 5 baud initialisation is used, then the 5 baud address shall be 33H and subsequent communication shall take place at 10 400 baud.

这里有四点关键信息是我们编写程序的时候必须要遵守的。

1. ISO14230-4 协议通信初始化有两种方式，分别是 5 波特率初始化和快速初始化，同时规定与 OBD 相关的 ECU 必须要支持这两种方式的其中一种。而我们设计诊断服务设备必须两种方式都支持。
2. ISO14230-4 协议的诊断服务工具还是汽车 ECU 都应该使用 Keyword 2025。
3. ISO14230-4 协议 5 波特率初始化使用的 5 波特率地址信息应该是 0x33。
4. ISO14230-4 协议数据通信的波特率是 10400。

这四点关键信息其中某些内容不理解不要紧，因为接下来很快就会做出系统性解释。首先看两种方式的初始化是怎么实现的。

-5 baud initialisation

这里的 5 波特率指的是发送地址 0x33 使用的波特率，ISO14230-4 使用 0x33 是遵循上述四点关键信息的第 3 点。使用 5 波特率发送 0x33，意味着发送一个码元使用的时间周期为 200 毫秒（即 $1s/5=0.2s$ ）。它的整个初始化过程如下图所示（截图源自 ISO14230-2 协议）。

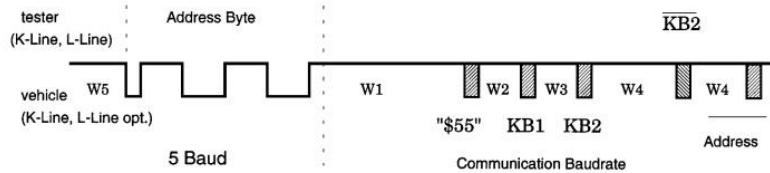


Figure 10 — 5 Baud initialization

Table 9 shows timing values for 5 Baud initialization. These are fixed values. They cannot be changed by the AccessCommunicationParameter service.

Table 9 — Timing values for 5 Baud initialization

Timing parameters	Values ms		Description
	min.	max.	
W1	60	300	Time from end of the address byte to start of synchronization pattern.
W2	5	20	Time from end of the synchronization pattern to the start of key byte 1.
W3	0	20	Time between key byte 1 and key byte 2.
W4	25	50	Time between key byte 2 (from the ECU) and its inversion from the tester. Also the time from the inverted key byte 2 from the tester and the inverted address from the ECU.
W5	300	—	Time before the tester starts to transmit the address byte.

Figure10 是整个初始化过程的时序图，Table 9 是它对应的时间值表格。初始化过程描述如下。Figure10 所示，在进行 5 波特率初始化的前，必须拉高 K 线和 L 线时间大于 300 毫秒 (W5)，然后 K 线和 L 线以波特率为 5 发送地址 (Address Byte) 0x33，发送完毕后拉高 L 线和 K 线，接下来的接收和发送均由 K 线进行，L 线停止工作，等待 60 毫秒到 300 毫秒 (W1) 获得 ECU 或者模拟器同步模式信息 (\$55)，这个 0x55 字节如果转换成二进制是 01010101 或者用示波器观察可以看到它的波形是连续的 8 位高低变化，它是发送给我们的诊断服务设备(C300 开发板)进行波特率计算的，也就是汽车通过这个 0x55 字节告诉诊断服务设备(C300 开发板) 正常通信的速率。而当前协议 ISO14230-4 不需要计算这个波特率，因为上述四点关键信息里第 4 点规定了当前协议必须是波特率 10400，我们在程序编写的时候直接设置波特率 10400 就行，没必要进行\$55 同步模式字节的计算，这有可能增加错误的风险。ECU 或者模拟器响应 0x55 同步模式字节后，间隔 5 到 20 毫秒 (W2) 响应 Keyword1 字节，间隔 0 到 20 毫秒 (W3) 响应 Keyword2 字节。Keyword 跟数据格式和时间间隔息息相关，所以它的具体解读在接下来的数据格式内具体讲述。Keyword1 和 Keyword2 响应完毕后，在 25 到 50 毫秒 (W4) 时间内诊断服务设备 (C300 开发板) 必须向 ECU 或者模拟器发送 Keyword2 按位取反后的字节。如果 ECU 或者模拟器接受了 Keyword2 按位取反后的字节在 5 到 50 毫秒(W4) 时间内会响应 0x33 按位取反字节 0xCC，则说明初始化成功。这时候就可以发送诊断服务数据请求 ECU 获得对应数据的响应，实现诊断服务功能。但是初始化成功即唤醒成功后，必须在 5 秒时间内有数据请求通信，如果超过 5 秒 K 线上无请求，K 线则会进入休眠状态，再次使用 K 线请求诊断服务就必须重新进行 5 波特率初始化过程。

上述过程为了直观展现，我利用 Neulen TBOX C300 开发板和 OBD 模拟器通信，利用分析仪采集的数据展示了上述激活过程如下图所示。



上图中前面三个 00，其实是 C300 开发板以 5 波特率发送 0x33 地址产生的，因为分析仪当前以 10400 进行采样，而 0x33 波特率是 5，采样点正好碰到是低电平，所以才出现 00 数据，这里我们知道它是在发送地址信息 0x33 就行了。接下来模拟器响应同步信息 0x55, Keyword1 是 0xE9, Keyword2 是 0x8F, C300 开发板发送 0x8F 取反字节 0x70 给模拟器，模拟器响应 0x33 取反响应给 C300 开发板表示初始化成功。初始化成功后必须在 5 秒内有通信请求，这里 C300 开发板向模拟器发送了数据流 Supported PIDs 请求。具体数据格式这里先搁置，在后面的数据格式里具体讲述。

-fast initialisation

快速初始化要比 5 波特率初始化简单很多，并且初始化的时间更短。整个初始化过程如下图所示（截图源自 ISO14230-2 协议）。

- first transmission after power on : $T_{idle} \geq W5min$;
- after completion of StopCommunication Service : $T_{idle} \geq P3min$;
- after stopping communication by timeout $P3max$: $T_{idle} \geq 0ms$.

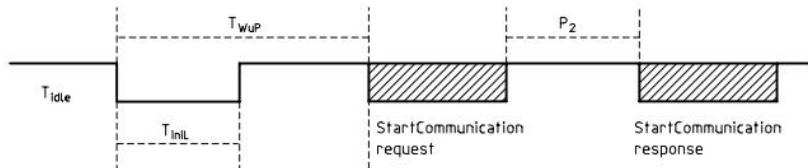


Figure 11 — Fast initialization

Table 10 — Timing values for fast initialization

Parameter	min value, ms	max value, ms	
T_{iniL}	25 ± 1 ms	24 ms	26 ms
T_{WuP}	50 ± 1 ms	49 ms	51 ms

Figure11 是整个初始化过程的时序图，Table 10 是它对应的时间值表格。初始化过程描述如下。Figure11 所示，对于快速初始化整个过程只有 K 线进行通信，L 线是不被使用的。初始化前，同样需要把 K 线拉高一定时间，如果是初次初始化拉高 K 线保持 300 毫秒 (W5) 以上。如果是通过停止通信服务 (SID=0x82) 请求停止服务的拉高 K 线保持 55 毫秒 (P3)。如果是通信超时休眠的，直接进入初始化过程即可。初始化过程拉低 K 线保持 25 毫秒 K 线低电平，然后拉高 K 线保持 25 毫秒高电平。然后进行开始通信请求 (SID=0x81)，如果在 25 到 50 毫秒 (P2) 之间获得汽车 ECU 或者模拟器的响应则说明初始化成功。初始化成功后也一样要保持 5 秒内有新的诊断请求，否则 K 线会进入休眠，需要重新初始化。

上述过程为了直观展现，我利用 Neulen TBOX C300 开发板和 OBD 模拟器通信，利用分析仪采集的数据展示了上述激活过程如下图所示。



第一个 00 字节一样是采样问题造成的, 利用 10400 波特率采样 25 毫秒的低电平只能采样到 00。C1 33 F1 81 66 就是开始通信请求数据, 81 是开始通信服务的 SID。图中获得了模拟器的积极响应, 响应数据是 83 F1 11 C1 E9 8F C4 , 其中 C1 是响应开始通信服务的 SID, E9 和 8F 分别代表 Keyword1 和 Keyword2。除此之外这两组数据头三个字节和最后一个字节分别代表什么意思, 这与 Keyword 息息相关。接下来在数据格式里具体讲述。

2. 数据格式

ISO14230-4 协议的数据格式和 Keyword 息息相关, 所以 ISO14230-4 协议也被称为 Keyword Protocol 2000 或者简称 KWP2000。下面我们看下 Keyword 如何决定数据格式和其它因素的。下面截图源自 ISO14230-2 协议, Figure 9 展示了 Keyword1 和 Keyword2 两个字节的每一位是如何传输的, 其实这就是串口传输方式, 每个字节从低位开始传输。

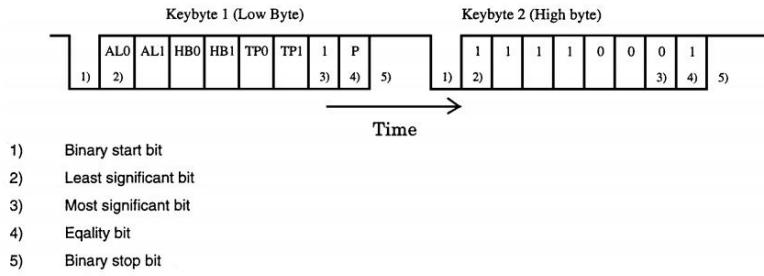


Figure 9 — Keybytes

Table 7 — Meaning of bit values in keybytes

Bit	Value	
	0	1
AL0	length inf. in format byte not supported	length inf. in format byte supported
AL1	add. length byte not supported	add. length byte supported
HBO	1 byte header not supported	1 byte header supported
HB1	Tgt/Src address in heater not supported	Tgt/Src address in heater supported
TP0 ¹⁾	normal timing parameter set	extended timing parameter set
TP1 ¹⁾	extented timing parameter set	normal timing parameter set

1) Only TP0, TP1 = 0,1 and 1,0 allowed.

这里我解释下 Table7 所罗列的每一位代表的意义。

Bit AL0 该位决定格式字节的低 6 位是否是表示 K 线数据帧长度。AL0=0 格式字节不表示数据长度；AL0=1 格式字节表示 K 线数据帧长度。这里顺便提一下，格式字节（Format byte）就是每一帧 K 线数据的第一个字节，它的高 2 位用来表示地址特性，低 6 位在 AL0=1 的情况下表示 K 线一帧长度。关于格式字节稍后会具体介绍。

Bit AL1 该位决定除了格式字节表示 K 线数据帧长度外，有没有增加别的字节表示 K 线数据帧长度。AL1=0 格式字节表示 K 线数据帧长度；AL1=1 时，如果格式字节低 6 位值是 0 就会增加别的字节表示 K 线数据帧长度，而如果格式字节低 6 位值不为 0，则该值表示 K 线数据帧长度。

Bit HBO 该位决定 K 线数据帧头部是否由一个字节即格式字节组成。HBO=0 K 线数据帧头部不是一个字节组成；HBO=1 K 线数据帧头部有一个字节组成。

Bit HB1 该位决定 K 线数据帧头部是否包含目标地址和源地址。HB1=0 K 线数据帧头部不包含目标地址和源地址；HB1=1 K 线数据帧头部包含目标地址和源地址。

Bit TP0 TP1 两位作为组合看待，因为这两位不能同时为 1 或者同时为 0，只会存在这两种情况（TP0=0 TP1=1）和（TP0=1 TP1=0）。这两位用于表示采用哪种定时参数设置，那么定时参数设置的值有 4 个分别是 P1, P2, P3, P4，分别代表的意义如下图 figure6 所示（截图源自 ISO14230-2 协议）。

Value	Description
P1	Inter byte time for ECU response
P2	Time between tester request and ECU response or two ECU responses
P3	Time between end of ECU responses and start of new tester request
P4	Inter byte time for tester request

Figure 6 — Timing

- P1 表示 ECU 响应数据的字节间时间间隔。
P2 表示诊断工具请求到 ECU 响应的时间。
P3 表示 ECU 结束响应到开始新的请求间隔时间。
P4 表示诊断设备请求数据字节间的时间间隔。

而上述的 P1, P2, P3, P4 具体值有两种时间表，分别是 Normal Timing Parameters Set 和 Extended Timing Parameters Set。如下图所示。

Table 3 — Normal Timing Parameters Set (for functional and physical addressing)

Values in milliseconds

Timing Parameter	Minimum values			Maximum values		
	Lower limit	Default	Resolution	Default	Upper limit	Resolution
P1	0	0	-	20	20	-
P2	0	25	0,5	50	See table 5	
P3	0	55	0,5	5 000	∞ (\$FF)	250
P4	0	5	0,5	20	20	-

Table 4 — Extended Timing Parameters Set (for physical addressing only)

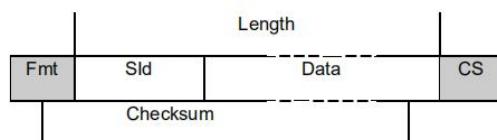
Values in milliseconds

Timing Parameter	Minimum values			Maximum values		
	Lower limit	Default	Resolution	Default	Upper limit	Resolution
P1	0	0	-	20	20	-
P2	0	25	0,5	50	See table 5	
P3	0	55	0,5	5 000	∞ (\$FF)	250
P4	0	5	0,5	20	20	-

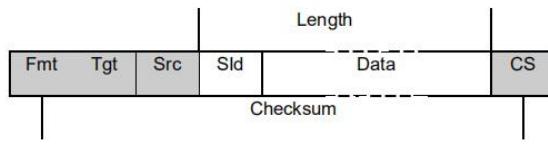
那么当 (TP0=0 TP1=1) 时，采用 Normal Timing Parameters Set (正常定时参数设置)；当 (TP0=1 TP1=0) 时，采用 Extended Timing Parameters Set (扩展定时参数设置)。

综上所述，Keyword 可以决定接下来通信的 K 线数据帧格式和采用的定时参数设置。下面根据 AL0, AL1, HBO, HB1 定义罗列四种 K 线数据帧格式。

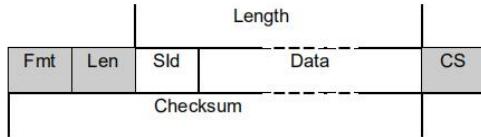
第一种，AL0=1, AL1=0, HBO=1, HB1=0 的 K 数据帧格式。



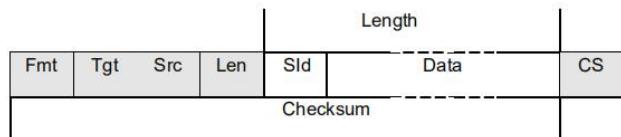
第二种，AL0=1, AL1=0, HBO=0, HB1=1 的 K 线数据帧格式。



第三种，AL0=0, AL1=1, HBO=1, HB1=0 的 K 线数据帧格式。



第四种，AL0=0, AL1=1, HBO=0, HB1=1 的 K 线数据帧格式。



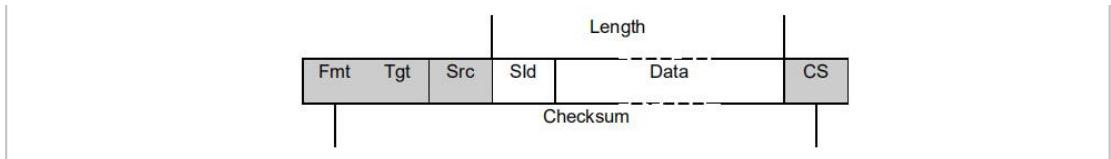
5. 那么 ISO14230-4 协议采用哪种 K 线数据帧格式和哪种定时参数设置呢？我们回顾前面讲解通信协议如何唤醒这部分内容提到的四点关键信息的第 2 点 **ISO14230-4 协议的诊断服务工具还是汽车 ECU 都应该使用 Keyword 2025**。Keyword 2025 表示的 Keyword 版本如下图红色方框所示。

Table 8 — Possible values of Keybytes

Keybytes		Supported			Time
Binary	Hex	Dec. 1)	Length information	Type of header	
KB2	KB1				
1000	1111	1101 0000	\$8FD0	2000	2)
1000	1111	1101 0101	\$8FD5	2005	format byte
1000	1111	1101 0110	\$8FD6	2006	additional length byte
1000	1111	0101 0111	\$8F57	2007	both modes possible
1000	1111	1101 1001	\$8FD9	2009	format byte
1000	1111	1101 1010	\$8FDA	2010	additional length byte
1000	1111	0101 1011	\$8F5B	2011	both modes possible
1000	1111	0101 1101	\$8F5D	2013	format byte
1000	1111	0101 1110	\$8F5E	2014	additional length byte
1000	1111	1101 1111	\$8FDF	2015	both mode possible
1000	1111	1110 0101	\$8FE5	2021	format byte
1000	1111	1110 0110	\$8FE6	2022	additional length byte
1000	1111	0110 0111	\$8F67	2023	both mode possible
1000	1111	1110 1001	\$8FE9	2025	format byte
1000	1111	1110 1010	\$8FEA	2026	additional length byte
1000	1111	0110 1011	\$8F6B	2027	both modes possible
1000	1111	0110 1101	\$8F6D	2029	format byte
1000	1111	0110 1110	\$8F6E	2030	additional length byte
1000	1111	1110 1111	\$8FEF	2031	both modes possible

其实 ISO14230-4 协议准确来说应该叫 KWP2025。那么它的 Keyword1=0xE9, Keyword2=0x8F。按二进制展开对应到 Table 7 里可以得到 AL0=1, AL1=0, HBO=0, HB1=1, TP0=0, TP1=1。也就是采用上述提到的第二种 K 线数据帧格式和使用正常定时参数设置。我把用到的 K 线数据帧格式和正常定时参数设置表格截图如下，之后的 ISO14230-4 协议解读和程序代码都必须遵守

如下截图。



ISO14230-4 采用的 K 线数据帧格式

Table 3 — Normal Timing Parameters Set (for functional and physical addressing)

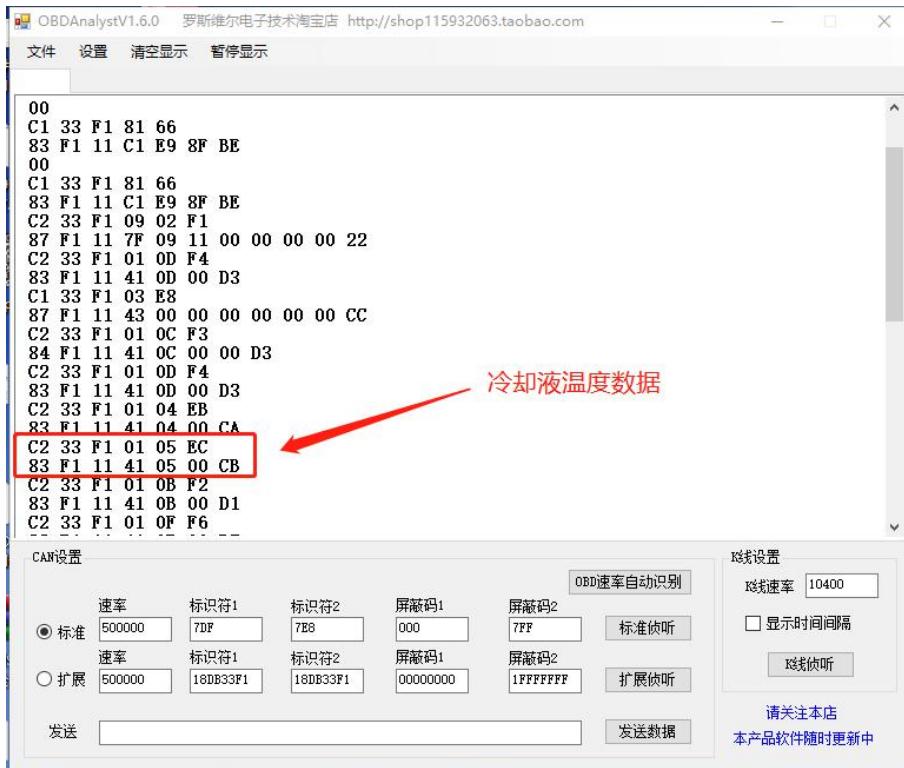
Values in milliseconds

Timing Parameter	Minimum values			Maximum values		
	Lower limit	Default	Resolution	Default	Upper limit	Resolution
P1	0	0	-	20	20	-
P2	0	25	0,5	50	See table 5	
P3	0	55	0,5	5 000	∞ (\$FF)	250
P4	0	5	0,5	20	20	-

ISO14230-4 采用的正常定时参数设置

其中 ISO14230-4 采用的 K 线数据帧格式的第一字节 Fmt 表示格式字节, Tgt 表示目标地址, Src 表示源地址, SId 和 Data 是属于应用层数据, 应用层数据 K 线传输就搭载在这个区域, 等会我们讲到的数据长度指的就是这部分数据长度, 不包括 Fmt, Tgt, Src 和 CS。CS 是累加校验字节。

下面我们实际举例展示 ISO14230-4 协议数据格式, 图中数据是利用 C300 开发板和 OBD 模拟器通信, OBD 分析仪采集到的实际数据。拿其中一组请求和响应数据进行分析, 红色方框表示发动机冷却液温度的原始通信数据, 包括发动机冷却液温度的诊断请求和响应。



发动机冷却液温度请求 K 线数据帧 C2 33 F1 01 05 EC

C2 表示 Fmt 格式字节，低 6 位值是 2 表示该帧承载有 2 个字节的应用层数据长度，即数据流 SID=0x01 和冷却液温度的 PID=0x05。

33 表示 Tgt 目标地址，同时 0x33 作为目标地址则意味着该帧命令属于功能性请求数据。

F1 表示 Src 源地址，0xF1 代表诊断服务工具的地址。

01 表示数据流 SID

05 表示冷却液温度的 PID

EC 表示校验字节，它是累加和校验字节，校验办法看下图计算器计算就明白。



发动机冷却液温度响应 K 线数据帧 83 F1 11 41 05 00 CB

83 表示 Fmt 格式字节, 低 6 位值是 3 表示承载应用层数据是 3 个字节即数据流响应
SID=0x41, 发动机冷却液温度 PID=0x05, 发动机冷却液温度值 0x00 (因为是模拟器模拟
没有调值, 00 对应-40 摄氏度)。

F1 表示 Tgt 目标地址, 目标地址是诊断服务设备, 所以是响应数据。

11 表示 Src 源地址, 表示该数据的来源。

41 表示数据流响应 SID。

05 表示发动机冷却液温度 PID。

00 表示发动机冷却液温度值, 具体定义将在接下来的应用层协议解读中介绍。

CB 表示校验字节。

这里细心的朋友可能会提出一个疑问, 为什么请求数据的格式字节是 0xC2, 以 0xCx 开头, 而
响应数据是 0x83, 以 0x8x 开头? 前面我也隐约提及格式字节的高 2 位是有特殊作用的, 如
下图所示 (截图源自 ISO14230-2 协议)。

Table 1 — Header message form

A1	A0	Mode
0	0	no address information
0	1	Exception mode (CARB)
1	0	with address information, physical addressing
1	1	with address information, functional addressing

这里的 A1 和 A0 分别表示格式字节的第 8 位最高位和第 7 位。请求数据以 0xCx 开头也就是
A1=1, A0=1 这时候表示功能性寻址, 也就是请求的是法规排放协议且目标地址是 0x33。响
应数据以 0x8x 开头 A1=1, A0=0 这时候表示物理性寻址。这和前面讲解的 ISO15765-4 协议概念
是类似的, 只是换了一种物理通信方式。

3. 通信信息管理

首先理解下定时时间参数设置, 前面我们已经知道 ISO14230-4 采用的是正常定时参数设置,
具体定时参数描述表格和定时参数设置表格再次截图如下所示。

Value	Description
P1	Inter byte time for ECU response
P2	Time between tester request and ECU response or two ECU responses
P3	Time between end of ECU responses and start of new tester request
P4	Inter byte time for tester request

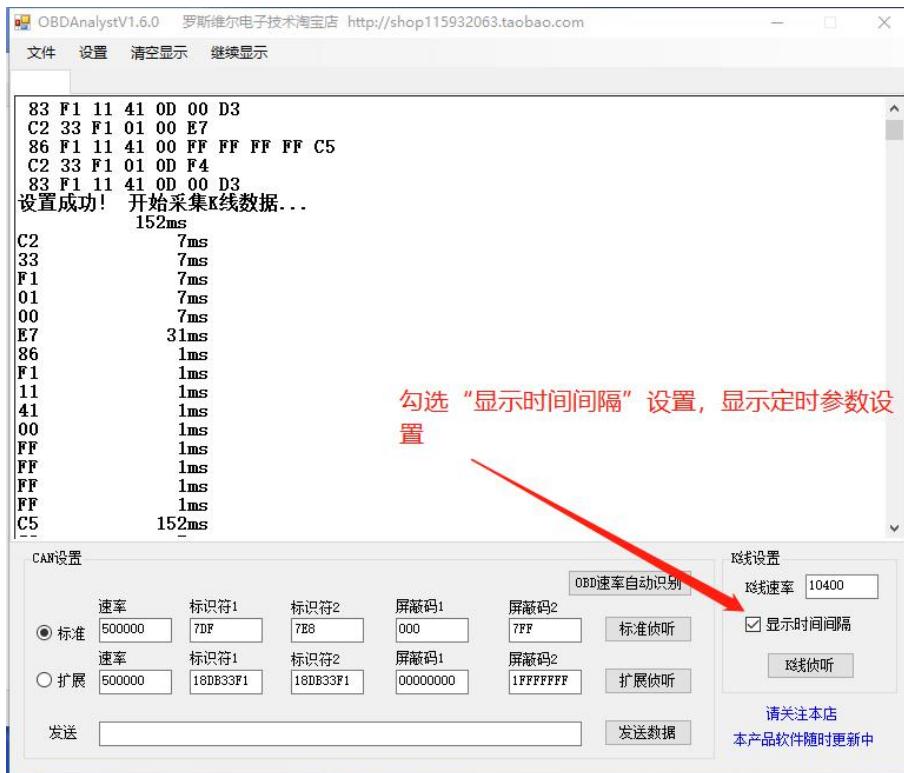
Figure 6 — Timing

Table 3 — Normal Timing Parameters Set (for functional and physical addressing)

Values in milliseconds

Timing Parameter	Minimum values			Maximum values		
	Lower limit	Default	Resolution	Default	Upper limit	Resolution
P1	0	0	-	20	20	-
P2	0	25	0,5	50	See table 5	
P3	0	55	0,5	5 000	∞ (\$FF)	250
P4	0	5	0,5	20	20	-

为了客观了解这两个表格在实际应用中的作用，我们使用 C300 开发板和 OBD 模拟器进行通信，当然 OBD 模拟器我们选择 ISO14230-4(fast init) 或者 ISO14230-4(5baud init) 中其中一个协议进行模拟，最后使用 OBD 分析仪勾选“显示时间间隔”选项，并点击【K 线侦听】按钮采集数据如下所示。



采集到的数据定时参数分析如下：

P1=1ms： P1 表示响应数据字节间时间间隔。我们的 OBD 模拟器采用 1MS, 协议中定义的正常定时参数设置要求是 0 到 20ms, 所以符合协议要求。

P2=31ms： P2 表示诊断服务设备发送请求数据等待 ECU 响应数据的时间，协议定义的正常

定时参数设置要求是 25ms 到 50ms, 我们的 OBD 模拟器响应时间是 31ms, 符合协议要求。

P3=152ms: P3 表示结束响应到新的诊断服务设备发送新请求信息的时间要求, 协议定义的正常定时参数设置要求是 55ms 到 5000ms, 符合协议要求。

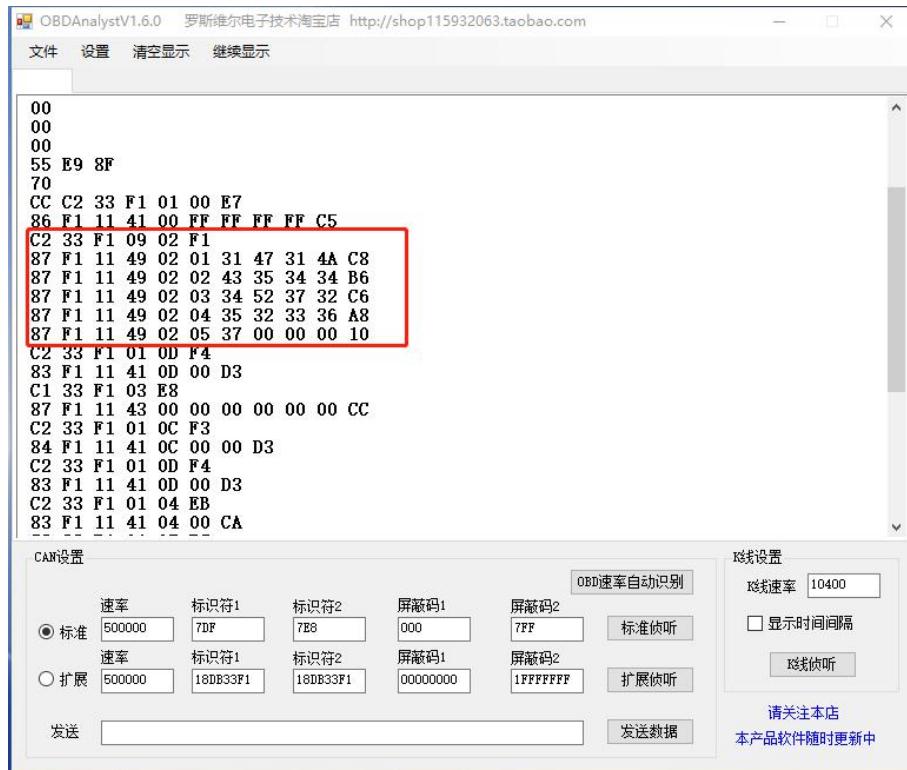
P4=7ms: P4 表示诊断服务设备请求数据字节间时间间隔。协议定义的正常定时参数设置要求是 5ms 到 20ms。符合协议要求。

这就是 KWP2000 的 K 线数据帧如何请求和响应的过程, 对于定时参数要求比较严格。K 线数据帧结构有没有固定长度要求呢? 对于 ISO14230-4 协议 KWP2000 的 K 线数据帧长度是有限制的, 长度限制要求在 ISO15031-5 应用层协议中有规定, 如下截图所示。

Table 13 — Diagnostic message format for ISO 9141-2, ISO 14230-4, SAE J1850

Header bytes (Hex)			Data bytes								
Priority/Type	Target address (hex)	Source address (hex)	#1	#2	#3	#4	#5	#6	#7	ERR	RESP
Diagnostic request at 10,4 kbit/s: SAE J1850 and ISO 9141-2											
68	6A	F1	Maximum 7 data bytes					Yes	No		
Diagnostic response at 10,4 kbit/s: SAE J1850 and ISO 9141-2											
48	6B	ECU addr	Maximum 7 data bytes					Yes	No		
Diagnostic request at 10,4 kbit/s (ISO 14230-4)											
11LL LLLLb	33	F1	Maximum 7 data bytes					Yes	No		
Diagnostic response at 10,4 kbit/s (ISO 14230-4)											
10LL LLLLb	F1	ECU addr	Maximum 7 data bytes					Yes	No		
Diagnostic request at 41,6 kbit/s (SAE J1850)											
61	6A	F1	Maximum 7 data bytes					Yes	Yes		
Diagnostic response at 41,6 kbit/s (SAE J1850)											
41	6B	ECU addr	Maximum 7 data bytes					Yes	Yes		

其中红色框表示 ISO14230-4 协议格式, 它的数据域 (Data bytes) 最大只能允许承载 7 个字节, 而这 7 个字节包括了应用层所有数据内容, 也就是包括的 SID PID 等服务信息, 而实际承载有效参数的字节是比较少的, 所以在读故障码或者车架号的时候一样需要多帧响应完成数据传输。为了客观说明, 我们用 OBD 分析仪采集 C300 开发板利用 ISO14230-4 协议读取车架号过程如下图所示。



红色方框 87 开头的 5 帧数据表示响应的车架号信息，其中真正的应用层原始数据是每帧响应数据 49 开始的 7 个字节，这 7 个字节中分别表示响应 SID, 信息类型, 信息计数器, 和 4 个字节的车架号信息，所以每帧只有 4 个字节是车架号的内容。应用层的定义我们将在下面的应用层如何定义的内容中再具体解读。

接下来我们再看下什么是积极响应 (Positive response) 和消极响应 (Negative response)，如果一个正常的请求数据获得 ECU 的积极响应，通常就是在响应数据中获得一个响应 SID 对应请求的数据中的 SID。如下图举例所示，积极响应的红色框里其实表示的是发动机转速，请求数据的 SID 是 0x01，响应 SID 是 0x41，这种情况被认为是积极响应，我们可以立刻在响应数据中获得正确的发动机转速信息。



再看消极响应红色框的举例，这是利用 OBD 模拟器设置无车架号的情况。C300 开发板请求车架号信息，SID 是 0x09，但是响应数据在该出现响应 SID=0x49 的位置出现了 7F，7F 就是消极响应的关键标志。消极响应数据结构如下图所示。

Table 11 — Negative response message format for ISO 14230-4, ISO 15765-4

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	Negative Response Service Identifier	M	7F	SIDNR
#2	Request Service Identifier	M	xx	SIDRQ
#3	ResponseCode	M	xx	RC_

7F 是关键标志，代表消极响应 SID，它替换积极响应 SID 的位置，下一个字节是请求 SID，用以表示对哪个服务进行消极响应，消极响应的举例中，读车架号获得了消极响应，所以这个位置是车辆信息的 SID=0x09。再下一个字节表示响应代码，理解为消极理由代码更贴切，举例中这个位置是 0x11，表示不支持该服务。响应代码说明消极理由的描述如下图所示。

Table 12 — Negative response code definition

Supported by ISO	Hex Value	Definition of Response Code	Mnemonic
14230-4	10	generalReject This response code indicates that the service is rejected but the server (ECU) does not specify the reason of the rejection.	GR
14230-4	11	serviceNotSupported This response code indicates that the requested action will not be taken because the server (ECU) does not support the requested service.	SNS
14230-4	12	subFunctionNotSupported-InvalidFormat This response code indicates that the requested action will not be taken because the server (ECU) does not support the arguments of the request message or the format of the argument bytes do not match the prescribed format for the specified service.	SFNSIF
14230-4 15765-4	21	busy-RepeatRequest This response code indicates that the server (ECU) is temporarily too busy to perform the requested operation. For ISO 15765-4 protocol the client (external test equipment) shall behave as defined in ISO 15765-4. In a multi-client (more than one external test equipment, e.g. telematic client) environment the diagnostic request message of one client might be blocked temporarily by a negative response message with response code \$21 while another client finishes a diagnostic task. Therefore this negative response code is only allowed to be used during the initialization sequence of the protocol. NOTE If the server (ECU) is able to perform the diagnostic task but needs additional time to finish the task and prepares the response message, the negative response message with response code \$78 are used instead of \$21.	BRR
14230-4 15765-4	22	conditionsNotCorrectOrRequestSequenceError This response code indicates that the requested action will not be taken because the server (ECU) prerequisite conditions are not met. This request may also occur when sequence-sensitive requests are issued in the wrong order.	CNCORSE
14230-4 15765-4	78	requestCorrectlyReceived-ResponsePending This response code indicates that the request message was received correctly, and that any parameters in the request message were valid, but the action to be performed may not be completed yet. This response code can be used to indicate that the request message was properly received and does not need to be re-transmitted, but the server (ECU) is not yet ready to receive another request. The negative response message with this response code may be repeated by the ECU(s) within $P_{2_{K-Line}} = P_{2_{CAN}} = P_{2_{max}}$ until the positive response message with the requested data is available.	RCR-RP

4. 应用层如何定义

ISO14230-4 协议应用层的定义和上一节 ISO15765-4 的应用层协议解读基本类似。只要是法规排放协议均使用 ISO15031-5 协议作为应用层和描述层协议。下面是 ISO14230-4 应用层协议 ISO15031-5 协议目录截图。

5.6 Format of data to be displayed	29
6 Diagnostic service definition for ISO 9141-2, ISO 14230-4, and SAE J1850	31
6.1 Service \$01 — Request current powertrain diagnostic data.....	31
6.2 Service \$02 — Request powertrain freeze frame data	35
6.3 Service \$03 — Request emission-related diagnostic trouble codes...	39
6.4 Service \$04 — Clear/reset emission-related diagnostic information	44
6.5 Service \$05 — Request oxygen sensor monitoring test results	46
6.6 Service \$06 — Request on-board monitoring test results for specific monitored systems.....	51
6.7 Service \$07 - Request emission-related diagnostic trouble codes detected during current or last completed driving cycle	56
6.8 Service \$08 — Request control of on-board system, test or component.....	57
6.9 Service \$09 — Request vehicle information	60
7 Diagnostic service definition for ISO 15765-4	73
7.1 Service \$01 — Request current powertrain diagnostic data.....	73
7.2 Service \$02 — Request powertrain freeze frame data	79
7.3 Service \$03 — Request emission-related diagnostic trouble codes...	84
7.4 Service \$04 — Clear/reset emission-related diagnostic information	87
7.5 Service \$05 — Request oxygen sensor monitoring test results	88
7.6 Service \$06 — Request on-board monitoring test results for specific monitored systems.....	89
7.7 Service \$07 - Request emission-related diagnostic trouble codes detected during current or last completed driving cycle.....	99
7.8 Service \$08 — Request control of on-board system, test or component.....	100
7.9 Service \$09 — Request vehicle information	104
Annex A (normative) PID (Parameter ID)/OBDMID (On-Board Monitor ID)/TID (Test ID)/INFOTYPE supported definition	114
Annex B (normative) PIDs (Parameter ID) for Services \$01 and \$02 scaling and definition.....	115

针对 C300 开发板第一版软件的需求，我们只讨论 Service \$01, \$03, \$09 这三个部分内容。

其它内容将在后续版本中继续讨论。

Service \$01 读当前动力总成数据（以下简称“数据流”）。在读数据流之前可以通过读 Supported PID 获得当前车辆对数据流的支持情况，这和 ISO15765-4 讨论的 Supported PID 基本类似，只是对于 ISO14230-4 协议的 K 线数据帧，只能一次查询一个 Supported PID，不像 ISO15765-4 协议可以一次查询所有 Supported PID 的功能。读 Supported PID 过程在 ISO15031-5 协议中有举例，如下图所示。

Table 20 — Request current powertrain diagnostic data request message

Message direction:	External test equipment → All ECUs		
Message Type:	Request		
Data Byte	Description (all values are in hexadecimal)	Byte Value (Hex)	Mnemonic
#1	Request current powertrain diagnostic data request SID	01	SIDRQ
#2	PID used to determine PID support for PIDs 01-20	00	PID

Table 21 — Request current powertrain diagnostic data response message

Message direction:	ECU#1 → External test equipment		
Message Type:	Response		
Data Byte	Description (all values are in hexadecimal)	Byte Value (Hex)	Mnemonic
#1	Request current powertrain diagnostic data response SID	41	SIDPR
#2	PID requested	00	PID
#3	Data byte A, representing support for PIDs 01, 03-08	10111111b = \$BF	DATA_A
#4	Data byte B, representing support for PIDs 09, 0B-10	10111111b = \$BF	DATA_B
#5	Data byte C, representing support for PIDs 11, 13, 15	10101000b = \$A8	DATA_C
#6	Data byte D, representing support for PIDs 19, 1C, 20	10010001b = \$91	DATA_D

Table 22 — Request current powertrain diagnostic data response message

Message direction:	ECU#2 → External test equipment		
Message Type:	Response		
Data Byte	Description (all values are in hexadecimal)	Byte Value (Hex)	Mnemonic
#1	Request current powertrain diagnostic data response SID	41	SIDPR
#2	PID requested	00	PID
#3	Data byte A, representing support for PID 01	10000000b = \$80	DATA_A
#4	Data byte B, representing support for PID 0D	00001000b = \$08	DATA_B
#5	Data byte C, representing no support for PIDs 11-18	00000000b = \$00	DATA_C
#6	Data byte D, representing no support for PIDs 19-20	00000000b = \$00	DATA_D

Table 23 — Request current powertrain diagnostic data request message

Message direction:	External test equipment → All ECUs		
Message Type:	Request		
Data Byte	Description (all values are in hexadecimal)	Byte Value (Hex)	Mnemonic
#1	Request current powertrain diagnostic data request SID	01	SIDRQ
#2	PID requested	20	PID

Table 24 — Request current powertrain diagnostic data response message

Message direction:	ECU#1 → External test equipment		
Message Type:	Response		
Data Byte	Description (all values are in hexadecimal)	Byte Value (Hex)	Mnemonic
#1	Request current powertrain diagnostic data response SID	41	SIDPR
#2	PID requested	20	PID
#3	Data byte A, representing support for PID 21	10000000b = \$80	DATA_A
#4	Data byte B, representing no support for PIDs 29-30	00000000b = \$00	DATA_B
#5	Data byte C, representing no support for PIDs 31-38	00000000b = \$00	DATA_C
#6	Data byte D, representing no support for PIDs 39-40	00000000b = \$00	DATA_D

上面的例子只是针对应用层进行举例，举例可以针对 ISO14230-4, ISO9141-2 和 SAEJ1850 三个协议。我们把例子套入 ISO14230-4 协议还原数据如下所示。

Table 20 请求当前动力总成数据请求信息（外部测试设备→所有 ECU）

C2 33 F1 01 00 E7

Table 21 请求当前动力总成数据响应信息（ECU#1→外部测试设备）

86 F1 11 41 00 BF BF A8 91 80

Table 22 请求当前动力总成数据响应信息（ECU#2→外部测试设备）

86 F1 12 41 00 80 08 00 00 52

Table 23 请求当前动力总成数据请求信息 Supported PID=0x20（外部测试设备→所有 ECU）

C2 33 F1 01 20 07

Table 24 请求当前动力总成数据响应信息（ECU#1→外部测试设备）

86 F1 11 41 20 80 00 00 00 69

也就是说，诊断服务设备发送诊断请求信息查询 Supported PID=0x00, 请求信息为 C2 33 F1 01 00 E7, 然后分别获得 ECU#1 (86 F1 11 41 00 BF BF A8 91 80) 和 ECU#2 (86 F1 12 41 00 80 08 00 00 52) 的数据响应。其中 ECU#1 中的 41 00 BF BF A8 91 代表应用层数据，41 是响应 SID, 00 是 Supported PID, BF BF A8 91 表示支持的 PID, 根据 ISO15031-5 协议中的 Annex A 定义所示（如下截图）。

Table A.1 — Supported PID/OBDMID/TID/INFOTYPE definition

Supported PID/OBDMID/TID/INFOTYPE (hex)	Scaling/bit Number of data bytes = 4 Data A - D or B - E: bit evaluation			External test equipment SI (Metric) / English display
	PID/OBDMID/TID/INFOTYPE supported (Hex)			
00	Data A bit 7 Data A bit 6 : Data D bit 0	01 02 : 20	0 = not supported 1 = supported	ISO 15031-4 specifies the behaviour of the external test equipment for how to interpret the data received to identify supported PIDs/OBDMIDs/TIDs/INFOTYPES for each ECU.
20	Data A bit 7 Data A bit 6 : Data D bit 0	21 22 : 40	0 = not supported 1 = supported	The ECU shall not respond to unsupported PID/TID/MID/InfoTypes ranges unless subsequent ranges have a supported

4个字节从高字节的最高位到低字节的最低位分别表示 PID=1 到 PID=0x20 的支持情况，其中对应的位置 1 表示支持，对应的位置 0 表示不支持，所以 BF BF A8 91 按照二进制展开 10111111 10111111 10101000 10010001, 1 表示支持，那么支持的 PID=0x01, 0x03-0x08, 0x09, 0x0B-0x10, 0x11, 0x13-0x15, 0x19, 0x1C-0x20。同理 ECU#2 中的 80 08 00 00 表示支持的 PID，那么支持的 PID 只有 0x01 和 0x0D。这时候对比 ECU#1 和 ECU#2 对 PID 的支持情况发现一个区别，ECU#1 支持的 PID 包括了 0x20，而 ECU#2 不支持 0x20。根据 S015031-5 协议中的 Annex A 定义，PID 为 0x00, 0x20, 0x40, 0x60, 0x80, 0xA0, 0xC0, 0xE0 都用于表示 Supported PID。ECU#1 明确支持 0x20 那就说明 0x20 后的 PID 还有支持，可通过发送 Supported PID=0x20 进行读取，所以例子中再次发送针对 Supported PID=0x20 的请求信息（C2 33 F1 01 20 07）获得 ECU#1 响应 86 F1 11 41 20 80 00 00 00 69，这时候可以看出 PID=0x21 是支持的。这个例子除了解释了 Supported PID 解析方法，同时告诉我们对于 ISO14230-4 协议读取 Supported PID 不能像 ISO15765-4 那样一次性读取，需要从 Supported PID=0x00 开始读，如果 Supported PID=0x00 查询到支持 0x20 再去读 Supported PID=0x20，以次类推，如果读到不支持下一个 Supported PID 就放弃读取。

接下来我们再看下，再知道 Supported PID 的情况下如何读取数据流。如下图所示，以 PID=0x19 为例。

Table 28 — Request current powertrain diagnostic data request message

Message direction:	External test equipment → All ECUs		
Message Type:	Request		
Data Byte	Description (all values are in hexadecimal)	Byte Value (Hex)	Mnemonic
#1	Request current powertrain diagnostic data request SID	01	SIDRQ
#2	PID: Oxygen Sensor Output Voltage (B2 - S2) Short Term Fuel Trim (B2 - S2)	19	PID

Table 29 — Request current powertrain diagnostic data response message

Message direction:	ECU#1 → External test equipment		
Message Type:	Response		
Data Byte	Description (all values are in hexadecimal)	Byte Value (Hex)	Mnemonic
#1	Request current powertrain diagnostic data response SID	41	SIDPR
#2	PID: Oxygen Sensor Output Voltage (B2 - S2) Short Term Fuel Trim (B2 - S2)	19	PID
#3	Oxygen Sensor Output Voltage (B2 - S2): 0,8 Volt	A0	DATA_A
#4	Short Term Fuel Trim (B2 - S2): 93,7 %	78	DATA_B

事实上在 ISO15031-5 协议里举了两个例子，但是我这里只拿其中一个说明问题。我把图中表格的数据按照 ISO14230-4 协议还原如下所示。

Table 28 请求当前动力总成诊断数据请求信息（外部测试设备→所有 ECU）

C2 33 F1 01 19 00

Table 29 请求当前动力总成诊断数据响应信息（ECU#1→外部测试设备）

84 F1 11 41 19 A0 78 F8

诊断服务设备发送当前动力总成诊断数据请求信息 PID=0x19 (C2 33 F1 01 19 00)，获得响应 (84 F1 11 41 19 A0 78 F8)，其中 41 表示数据流响应 SID, 19 表示氧传感器输出电压和短期燃油修正的 PID, A0 78 具体数据流参数，A0 对应氧传感器输出电压 (B2-S2), 78

对应短期燃油修正 (B2-S2)，他们由原始值转换成实际值的转换关系如下图所示，其转换关系在协议中被称为分辨率 (Scaling)。

Table B.21 — PID \$14 - \$1B definition

PID (hex)	Description Use if PID \$13 is supported!	Data byte	Min. value	Max. value	Scaling/bit	External test equipment SI (Metric) / English display
14	Bank 1 – Sensor 1	These PIDs shall be used for a conventional, 0 to 1 volt oxygen sensor. Any sensor with a different full scale value shall be normalized to provide nominal full scale at \$C8 (200 decimal). Wide-range/linear oxygen sensors shall use PIDs \$24 to \$2B or PIDs \$34 to \$3B.				
15	Bank 1 – Sensor 2					
16	Bank 1 – Sensor 3					
17	Bank 1 – Sensor 4					
18	Bank 2 – Sensor 1					
19	Bank 2 – Sensor 2					
1A	Bank 2 – Sensor 3					
1B	Bank 2 – Sensor 4					
	Oxygen Sensor Output Voltage (Bx-Sy)	A	0 V	1,275 V	0,005 V	O2Sxy: x.xxx V
	Short Term Fuel Trim (Bx-Sy) (associated with this sensor \$FF if this sensor is not used in the calculation)	B	- 100,00 % (lean)	99,22 % (rich)	100/128 % (0 % at 128)	SHRTFTxy: xxx.x %

可以了解到 PID=0x14 到 PID=0x1B，它们表示的都是氧传感器输出电压和短期燃油修正。只是传感器位置不同。回到例子中的 PID=0x19 表示第 2 组的第 2 个传感器。图中可以了解到氧传感器输出电压分辨率是每比特表示 0.005V，那么把 0xA0 代入计算，0xA0*0.005=0.8，所以举例中氧传感器输出电压 (B2-S2) 0.8V; 短期燃油修正分辨率是每比特 100/128，将例子中的 0x78 代入计算 0x78*100/128=93.75。所以举例中的短期燃油修正 (B2-S2) 93.75%。

Service \$03 读与排放相关的诊断故障码（以下简称“读故障码”）。关于故障码结构，因为前面章节已经做了详细解读，这里不再赘述，可以参看 7.2.1 节 ISO15765-4 协议解读的 220 页和 221 页。下面我将按照 ISO15031-5 协议举例讲述读故障码的过程。如下截图所示。

Table 42 — Request current powertrain diagnostic data request message

Message direction:	External test equipment → All ECUs		
Message Type:	Request		
Data Byte	Description (all values are in hexadecimal)	Byte Value (Hex)	Mnemonic
#1	Request current powertrain diagnostic data request SID	01	SIDRQ
#2	PID: Number of emission-related DTCs and MIL status	01	PID

Table 43 — Request current powertrain diagnostic data response message

Message direction:	ECU#1 → External test equipment		
Message Type:	Response		
Data Byte	Description (all values are in hexadecimal)	Byte Value (Hex)	Mnemonic
#1	Request current powertrain diagnostic data response SID	41	SIDPR
#2	PID: Number of emission-related DTCs and MIL status	01	PID
#3	MIL: ON; Number of emission-related DTCs: 06	86	DATA_A
#4	Misfire -, Fuel system -, Comprehensive monitoring	33	DATA_B
#5	Catalyst -, Heated catalyst -, ..., monitoring supported	FF	DATA_C
#6	Catalyst -, Heated catalyst -, ..., monitoring test complete/not complete	63	DATA_D

Table 44 — Request current powertrain diagnostic data response message

Message direction:	ECU#2 → External test equipment		
Message Type:	Response		
Data Byte	Description (all values are in hexadecimal)	Byte Value (Hex)	Mnemonic
#1	Request current powertrain diagnostic data response SID	41	SIDPR
#2	PID: Number of emission-related DTCs and MIL status	01	PID
#3	MIL: OFF; Number of emission-related DTCs: 01	01	DATA_A
#4	Comprehensive monitoring: supported, test complete	44	DATA_B
#5	Catalyst -, Heated catalyst -, ..., monitoring supported	00	DATA_C
#6	Catalyst -, Heated catalyst -, ..., monitoring test complete/not complete	00	DATA_D

Table 45 — Request current powertrain diagnostic data response message

Message direction:	ECU#3 → External test equipment		
Message Type:	Response		
Data Byte	Description (all values are in hexadecimal)	Byte Value (Hex)	Mnemonic
#1	Request current powertrain diagnostic data response SID	41	SIDPR
#2	PID: Number of emission-related DTCs and MIL status	01	PID
#3	MIL: OFF; Number of emission-related DTCs: 00	00	DATA_A
#4	Comprehensive monitoring: supported, test complete	00	DATA_B
#5	Catalyst -, Heated catalyst -, ..., monitoring supported	00	DATA_C
#6	Catalyst -, Heated catalyst -, ..., monitoring test complete/not complete	00	DATA_D

Table 46 — Request emission-related diagnostic trouble codes request message

Message direction:	External test equipment → All ECUs		
Message Type:	Request		
Data Byte	Description (all values are in hexadecimal)	Byte Value (Hex)	Mnemonic
#1	Request emission-related DTC request SID	03	SIDRQ

Table 47 — Request emission-related diagnostic trouble codes response message

Message direction:	ECU #1 → External test equipment		
Message Type:	Response		
Data Byte	Description (all values are in hexadecimal)	Byte Value (Hex)	Mnemonic
#1	Request emission-related DTC response SID	43	SIDPR
#2	DTC#1 High Byte of P0143	01	DTC1HI
#3	DTC#1 Low Byte of P0143	43	DTC1LO
#4	DTC#2 High Byte of P0196	01	DTC2HI
#5	DTC#2 Low Byte of P0196	96	DTC2LO
#6	DTC#3 High Byte of P0234	02	DTC3HI
#7	DTC#3 Low Byte of P0234	34	DTC3LO

Table 48 — Request emission-related diagnostic trouble codes response message

Message direction:	ECU #2 → External test equipment		
Message Type:	Response		
Data Byte	Description (all values are in hexadecimal)	Byte Value (Hex)	Mnemonic
#1	Request emission-related DTC response SID	43	SIDPR
#2	DTC#1 High Byte of P0443	04	DTC1HI
#3	DTC#1 Low Byte of P0443	43	DTC1LO
#4	DTC#2 High Byte: 00	00	DTC2HI
#5	DTC#2 Low Byte: 00	00	DTC2LO
#6	DTC#3 High Byte: 00	00	DTC3HI
#7	DTC#3 Low Byte: 00	00	DTC3LO

Table 49 — Request emission-related diagnostic trouble codes response message

Message direction:	ECU #1 → External test equipment		
Message Type:	Response		
Data Byte	Description (all values are in hexadecimal)	Byte Value (Hex)	Mnemonic
#1	Request emission-related DTC response SID	43	SIDPR
#2	DTC#1 High Byte of P02CD	02	DTC1HI
#3	DTC#1 Low Byte of P02CD	CD	DTC1LO
#4	DTC#2 High Byte of P0357	03	DTC2HI
#5	DTC#2 Low Byte of P0357	57	DTC2LO
#6	DTC#3 High Byte of P0A24	0A	DTC3HI
#7	DTC#3 Low Byte of P0A24	24	DTC3LO

Table 50 — Request emission-related diagnostic trouble codes response message

Message direction:	ECU #3 → External test equipment		
Message Type:	Response		
Data Byte	Description (all values are in hexadecimal)	Byte Value (Hex)	Mnemonic
#1	Request emission-related DTC response SID	43	SIDPR
#2	DTC#1 High Byte: 00	00	DTC1HI
#3	DTC#1 Low Byte: 00	00	DTC1LO
#4	DTC#2 High Byte: 00	00	DTC2HI
#5	DTC#2 Low Byte: 00	00	DTC2LO
#6	DTC#3 High Byte: 00	00	DTC3HI
#7	DTC#3 Low Byte: 00	00	DTC3LO

因为协议中的举例是兼容 ISO9141-2, ISO14230-4, J1850 协议的, 所以它只展现应用层部分, 下面我把 Table42 到 Table50 根据 ISO14230-4 协议进行还原如下所示。

Table 42 请求当前动力总成诊断数据请求信息 (外部测试设备→所有 ECU)

C2 33 F1 01 01 E8

Table 43 请求当前动力总成诊断数据响应信息 (ECU#1→外部测试设备)

86 F1 11 41 01 86 33 FF 63 E5

Table 44 请求当前动力总成诊断数据响应信息 (ECU#2→外部测试设备)

86 F1 12 41 01 01 44 00 00 10

Table 45 请求当前动力总成诊断数据响应信息 (ECU#3→外部测试设备)

86 F1 13 41 01 00 00 00 00 CC

Table 46 请求与排放相关的诊断故障码请求信息 (外部测试设备→所有 ECU)

C1 33 F1 03 E8

Table 47 请求与排放相关的诊断故障码响应信息 (ECU#1→外部测试设备)

87 F1 11 43 01 43 01 96 02 34 DD

Table 48 请求与排放相关的诊断故障码响应信息 (ECU#2→外部测试设备)

87 F1 12 43 04 43 00 00 00 00 14

Table 49 请求与排放相关的诊断故障码响应信息 (ECU#1→外部测试设备)

87 F1 11 43 02 CD 03 57 0A 24 23

Table 50 请求与排放相关的诊断故障码响应信息 (ECU#3→外部测试设备)

87 F1 13 43 00 00 00 00 00 00 CE

我们来描述下这个过程，诊断测试设备如C300开发板发送数据流请求信息 C2 33 F1 01 01 E8 获得ECU#1响应 86 F1 11 41 01 86 33 FF 63 E5，ECU#2响应 86 F1 12 41 01 01 44 00 00 10 和ECU#3响应 86 F1 13 41 01 00 00 00 00 00 CC。这是协议推荐给我们的一种读取故障码的方法，通过预先读数据流中的故障码个数确认当前系统所存在的故障码数量。这个故障码个数的转换关系如下图所示。

Table B.2 — PID \$01 definition

PID (hex)	Description	Data byte	Scaling/bit	External test equipment SI (Metric) / English display
01	Monitor status since DTCs cleared The bits in this PID shall report two pieces of information for each monitor: — monitor status since DTCs were last cleared, saved in NVRAM or Keep Alive RAM; and — monitors supported on this vehicle.			
	Number of emission-related DTCs and MIL status	A (bit)	byte 1 of 4	DTC and MIL status:
	# of DTCs stored in this ECU	0-6	hex to decimal	DTC_CNT: xx d
	Malfunction Indicator Lamp (MIL) Status	7	0 = MIL OFF; 1 = MIL ON	MIL: OFF or ON

也就是把响应数据参数的第一字节 A 字节的低 7 位转换成十进制便是故障码个数。从以上的ECU响应数据得知，ECU#1有6个故障码，ECU#2有1个故障码，ECU#3没有故障码。接下来诊断测试设备发送故障码请求信息 C1 33 F1 03 E8，

获得ECU#1响应：

87 F1 11 43 01 43 01 96 02 34 DD

87 F1 11 43 02 CD 03 57 0A 24 23

ECU#2响应：

87 F1 12 43 04 43 00 00 00 00 14

ECU#3响应：

87 F1 13 43 00 00 00 00 00 00 CE

响应的每帧KWP2000数据的第5个字节开始每两个字节表示一个故障码，所以得知

ECU#1故障码是P0143, P0196, P0234, P02CD, P0357, P0A24

ECU#2故障码是P0443

ECU#3无故障码

Service \$09 读车辆信息，车辆信息包含很多内容，我们当前第一版软件只做了读车架号这一项，其它功能将在后续版本软件再进一步讨论。读车辆信息有一个功能支持查询叫“支持信息类型（Supported InfoType）”，类似于数据流的 Supported PID，请求和响应的方式也基本一致，只是服务字节为车辆信息的服务字节 0x09, 后跟 PID 的位置叫做信息类型（InfoType）。具体请求和响应结构如下图所示。

6.9.2.1 Request vehicle information request message definition (read supported InfoType)

Table 86 — Request vehicle information request message (read supported InfoType)

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	Request vehicle information request SID	M	09	SIDRQ
#2	InfoType (see Annex A)	M	xx	INF_TYP

6.9.2.2 Request vehicle information response message definition (report supported InfoType)

Table 87 — Request vehicle information response message (report supported InfoType)

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	Request vehicle information response SID	M	49	SIDPR
#2	InfoType	M	xx	INF_TYP_
#3	MessageCount	M	xx	MC_
#4	data record of InfoType = [Data A: supported InfoTypes, Data B: supported InfoTypes, Data C: supported InfoTypes, Data D: supported InfoTypes]	M	xx	DATA_REC_- DATA_A
#5		M	xx	DATA_B
#6		M	xx	DATA_C
#7		M	xx	DATA_D

车架号的信息类型（InfoType）是 0x02, 如果要查询车架号是否支持，那么上图 Table 86 请求信息的信息类型（InfoType）用 0x00。因为根据 Annex A 表格定义 Supported InfoType=0x00 可查询 0x01 到 0x20 的信息类型（InfoType），所以只要判断响应信息即上图的 Table 87 report supported InfoType 的 Data A 字节的第 6 位（从 0 位开始算）是 1 就表示支持车架号，否则表示不支持。下面根据上述的过程利用 ISO14230-4 协议还原 KWP2000 数据格式展示 Supported InfoType 请求和响应过程。

Table 86 请求车辆信息请求信息（读被支持的信息类型）

C2 33 F1 09 00 EF

Table 87 请求车辆信息响应信息（被支持的信息类型报告）

87 F1 11 49 00 01 40 00 00 00 13

我虚构了 Table 87 响应数据 A 字节为 0x40, 其它字节均为 0, 表示该 ECU 只支持读车架号信息，现实很少存在这种情况。我只是为了说明只要是 A 字节的第 6 位为 1 则表示可以支持读车架号信息。下面我们看如何利用 ISO14230-4 即 KWP2000 读取车架号。协议举例中的车架号是 1G1JC5444R7252367。下面是协议举例截图。

Table 92 — Request vehicle information request message

Message direction:	External test equipment → All ECUs		
Message Type:	Request		
Data Byte	Description (all values are in hexadecimal)	Byte Value (Hex)	Mnemonic
#1	Request vehicle information request SID	09	SIDRQ
#2	InfoType: VIN	02	INF_TYP

Table 93 — Request vehicle information response message (1)

Message direction:	ECU#1 → External test equipment		
Message Type:	Response		
Data Byte	Description (all values are in hexadecimal)	Byte Value (Hex)	Mnemonic
#1	Request vehicle information response SID	49	SIDPR
#2	InfoType: VIN	02	INFTYP
#3	MessageCount VIN = 1 st response message	01	MC_VIN
#4	Data A: Fill byte	00	DATA_A
#5	Data B: Fill byte	00	DATA_B
#6	Data C: Fill byte	00	DATA_C
#7	Data D: '1'	31	DATA_D

Table 94 — Request vehicle information response message (2)

Message direction:	ECU#1 → External test equipment		
Message Type:	Response		
Data Byte	Description (all values are in hexadecimal)	Byte Value (Hex)	Mnemonic
#1	Request vehicle information response SID	49	SIDPR
#2	InfoType: VIN	02	INFTYP
#3	MessageCount VIN = 2 nd response message	02	MC_VIN
#4	Data A: 'G'	47	DATA_A
#5	Data B: '1'	31	DATA_B
#6	Data C: 'J'	4A	DATA_C
#7	Data D: 'C'	43	DATA_D

Table 95 — Request vehicle information response message (3)

Message direction:	ECU#1 → External test equipment		
Message Type:	Response		
Data Byte	Description (all values are in hexadecimal)	Byte Value (Hex)	Mnemonic
#1	Request vehicle information response SID	49	SIDPR
#2	InfoType: VIN	02	INFTYP
#3	MessageCount VIN = 3 rd response message	03	MC_VIN
#4	Data A: '5'	35	DATA_A
#5	Data B: '4'	34	DATA_B
#6	Data C: '4'	34	DATA_C
#7	Data D: '4'	34	DATA_D

Table 96 — Request vehicle information response message (4)

Message direction:	ECU#1 → External test equipment		
Message Type:	Response		
Data Byte	Description (all values are in hexadecimal)	Byte Value (Hex)	Mnemonic
#1	Request vehicle information response SID	49	SIDPR
#2	InfoType: VIN	02	INFTYP
#3	MessageCount VIN = 4 th response message	04	MC_VIN
#4	Data A: 'R'	52	DATA_A
#5	Data B: '7'	37	DATA_B
#6	Data C: '2'	32	DATA_C
#7	Data D: '5'	35	DATA_D

Table 97 — Request vehicle information response message (5)

Message direction:	ECU#1 → External test equipment		
Message Type:	Response		
Data Byte	Description (all values are in hexadecimal)	Byte Value (Hex)	Mnemonic
#1	Request vehicle information response SID	49	SIDPR
#2	InfoType: VIN	02	INF_TYP
#3	MessageCount VIN = 5 th response message	05	MC_VIN
#4	Data A: '2'	32	DATA_A
#5	Data B: '3'	33	DATA_B
#6	Data C: '6'	36	DATA_C
#7	Data D: '7'	37	DATA_D

利用 ISO14230-4 协议将举例的 Table92 到 Table97 数据还原如下所示。

Table 92 请求车辆信息请求信息（读车架号）

C2 33 F1 09 02 F1

Table 93 请求车辆信息响应信息（1）

87 F1 11 49 02 01 00 00 00 31 06

Table 94 请求车辆信息响应信息（2）

87 F1 11 49 02 02 47 31 4A 43 3B

Table 95 请求车辆信息响应信息（3）

87 F1 11 49 02 03 35 34 34 34 A8

Table 96 请求车辆信息响应信息（4）

87 F1 11 49 02 04 52 37 32 35 C8

Table 97 请求车辆信息响应信息（5）

87 F1 11 49 02 05 32 33 36 37 AB

这是举例的数据，现实中我们发现实际采集的数据和例子中的有些差别，也就是 Table93 中的填充字节（Fill Byte）被放在车架号信息的最后。所以开发中要根据实际情况做好调整。读取车架号的过程，首先诊断服务设备发送请求数据 C2 33 F1 09 02 F1，获得 ECU 响应 5 帧数据，这 5 帧数据并没有握手和确认过程，编写程序的时候一定要注意在 P2 定时参数内抓取并存储数据。每帧数据只有字节 7 到字节 10 这四个字节是车架号信息，它们以 ASCII 编码形式表示车架号信息。

7.3.2 ISO14230-4 协议在 Neulen TBOX C300 开发板中的程序实现

首先看 ISO14230-4 初始化函数如何实现，下面分别介绍 **5 baud initialisation** 和 **fast initialisation** 这两种初始化过程实现代码。

5 baud initialisation 实现代码如下图所示

```

~ 27 /* ****
28 * @描述: ISO14230-4地址激活唤醒判断
29 * @参数: NONE
30 * @返回值: NL_OK:支持 NL_NOK:不支持
31 *****/
32 NLStatus ISO14230_4ADDR_WakeUp(void)
33 {
34     uint8_t i;
35     NLStatus err;
36     if(NL_OBD_Wakeup_Addr(0x33) == NL_OK)
37     {
38         err = NL_NOK;
39         for(i = 0; i < 3; i++)
40         {
41             NL_OBD_SendKLINEFrame(LinkCmd14230,Single,800,&err);
42             NL_Delay(1500);
43             if(err == NL_OK)
44             {
45                 break;
46             }
47         }
48         return err;
49     }
50     return NL_NOK;
51 }
52
53

```

其中 36 行 `NL_OBD_Wakeup_Addr` 函数是我们将要重点介绍的函数，它完成了 **5 baud initialisation** 的关键过程，包括发送 0x33 地址信息，获得汽车 ECU 同步参数，`keyword1`，`keyword2` 并且完成相应的握手，下面我们会做详细介绍。在这些过程完成之后 39 到 47 行发送链路保持请求数据，但是当前版本软件中这个链路请求数据其实是数据流的 support PID 请求数据。请求如果获得汽车 ECU 响应数据才能证明 **5 baud initialisation** 成功。值得注意的是，这里使用了 `for` 循环 3 次。这是我们在实车测试过程中根据实际测试进行添加的。在一些相对老的车型，在 36 行代码成功执行完毕后，发送一次链路保持请求数据往往不会得到汽车 ECU 任何响应，需要重复发送请求 2 次到 3 次才能获得响应。43 行到 46 行，如果变量 `err` 值是 `NL_OK`，则说明获得了汽车 ECU 的响应数据，通过 45 行的 `break` 语句退出循环，在 48 行整个函数返回 `NL_OK` 说明初始化成功。否则 3 次循环完毕未获得响应或者 `NL_OBD_Wakeup_Addr` 函数返回 `NL_NOK`，则通过 51 行让函数返回 `NL_NOK`。下面我们具体看 36 行调用的 `NL_OBD_Wakeup_Addr` 函数是如何实现的，看下面代码截图。

```

150  */
151  * @描述: 5 baud 初始化
152  * @参数: addr 地址码
153  * @返回值: NL_OK 初始化成功 NL_NOK 初始化失败
154  ****
155  NLStatus NL_OBD_Wakeup_Addr(uint8_t addr)
156  {
157      uint32_t i;
158      uint8_t data;
159      NLStatus re;
160
161      _NL_OBD_KLINE_SwPinMode(OUTPUT_PP);
162      _NL_OBD_KLINE_ENABLE();
163      _NL_OBD_LLINE_ENABLE();
164      _NL_Delay(2000);
165      _NL_OBD_KLINE_LOW();
166      _NL_Delay(200);
167      for (i = 0; i < 8; i++)
168      {
169          if ((addr >> i) & 0x01) != 0
170          {
171              _NL_OBD_KLINE_HIGHT();
172          }
173          else
174          {
175              _NL_OBD_KLINE_LOW();
176          }
177          _NL_Delay(200);
178      }
179      _NL_OBD_LLINE_DISABLE();
180      _NL_OBD_KLINE_HIGHT();
181      _NL_OBD_KLINE_SwPinMode(AF_PP);
182      OBDFlag.RxFlag = NL_FAILURE;
183      _NL_OBD_SendKLINEFrame_Init();
184      for(i = 0; i < 800; i++)// 延时800MS 协议最大值是300MS
185      {
186          _NL_Delay(1);
187          if(OBDFlag.RxFlag == NL_SUCCESS) break;
188      }
189
190      if(OBDFlag.RxFlag == NL_SUCCESS)
191      {
192          _NL_Delay(35);
193          OBDFlag.RxFlag = NL_FAILURE;
194          data = ~KLINERxRAM.RxRAM[0][2];
195          KLINE_SendByte(data);
196          for(i = 0; i < 800; i++)// 延时800MS 协议最大值是50MS
197          {
198              _NL_Delay(1);
199              if(OBDFlag.RxFlag == NL_SUCCESS) break;
200          }
201          data = ~addr;
202          if(OBDFlag.RxFlag == NL_SUCCESS && KLINERxRAM.RxRAM[0][4] == data)
203          {
204              re = NL_OK;
205          }
206          else
207          {
208              re = NL_NOK;
209          }
210      }
211      else
212      {
213          re = NL_NOK;
214      }
215      _NL_OBD_KLINE_HIGHT();
216      _NL_OBD_KLINE_DISABLE();
217  }

```

这个函数完全按照协议描述进行编写的，在 7.3.1 IS014230-4 协议解读 中 -5 baud initialisation 描述中使用到协议文档一个截图如下所示。

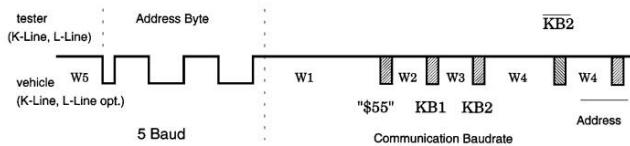


Figure 10 — 5 Baud initialization

Table 9 shows timing values for 5 Baud initialization. These are fixed values. They cannot be changed by the AccessCommunicationParameter service.

Table 9 — Timing values for 5 Baud initialization

Timing parameters	Values ms		Description
	min.	max.	
W1	60	300	Time from end of the address byte to start of synchronization pattern.
W2	5	20	Time from end of the synchronization pattern to the start of key byte 1.
W3	0	20	Time between key byte 1 and key byte 2.
W4	25	50	Time between key byte 2 (from the ECU) and its inversion from the tester. Also the time from the inverted key byte 2 from the tester and the inverted address from the ECU.
W5	300	—	Time before the tester starts to transmit the address byte.

现在我们可以套用这个截图理解 NL_OBD_Wakeup_Addr 函数代码。161 行设置 K 线对应串口发送引脚 TX 为普通引脚的推挽式输出模式，这是为了发送 5baud 地址信息准备。162 行和 163 行分别使能 K 线和 L 线通信，发送 5baud 地址信息的时候测试设备需要 L 线参与。164 行延时 2000 毫秒可把这个参数最低设置为 300 毫秒，此延时语句对应上图时序的 W5 定时值。165 行和 166 行拉低 K 线和 L 线 200 毫秒作为 5baud 地址信息的起始位。167 行到 178 行以 5 波特率的速率发送 0x33 地址信息，当然此时 0x33 作为函数参数传入。地址信息发送完毕后，179 行关闭 L 线，180 行让 K 线保持高电平。181 行恢复 K 线对应串口 TX 引脚作为串口复用功能引脚。182 行和 183 行对即将用到的串口中断及变量进行初始化，以等待汽车 ECU 响应。184 行到 188 行等待汽车 ECU 响应同步参数，keyword1 和 keyword2，这个等待时间对应上图的 W1 定时值，程序中等待了 800 毫秒，协议中要求 300 毫秒即可。如果在 W1 时间内接收到同步参数，keyword1 和 keyword2，OBDFlag.RxFlag 的值为 NL_SUCCESS，否则为 NL_FAILURE。NL_FAILURE 时函数将 212 行，214 行和 215 行代码，拉高 K 线，失能 K 线并让函数返回 NL_NOK 表示初始化失败。189 行判断 OBDFlag.RxFlag 的值为 NL_SUCCESS 时，将继续初始化过程，此时汽车 ECU 响应的同步参数，keyword1 和 keyword2 被分别存储在 KLINERxRAM.RxRAM[0][0]，KLINERxRAM.RxRAM[0][1] 和 KLINERxRAM.RxRAM[0][2] 中。193 行和 194 行取 keyword2 即上图的 KB2 值存储在 KLINERxRAM.RxRAM[0][2] 中取反发送给汽车 ECU 以表示 C300 开发板接收到相关信息。195 行到 199 行等待时间即上图的 W4，程序中等待 800 毫秒，协议规定 50 毫秒即可。在这个时间内汽车 ECU 会响应地址信息 0x33 取反字节 0xCC，所以程序中的 200 行和 201 行就是比较 ECU 响应的是不是 0xCC，它会被中断存储在 KLINERxRAM.RxRAM[0][4] 中，如果响应正确函数便完成初始化过程，函数返回 NL_OK。

fast initialisation 实现代码如下所示。

```

54  */
55  * @描述: ISO14230-4快速唤醒判断
56  * @参数: NONE
57  * @返回值: NL_OK:支持 NL_NOK:不支持
58  ****
59  NLStatus ISO14230_4HL_WakeUp(void)
60  {
61      uint8_t i;
62      NLStatus err;
63      err = NL_NOK;
64      for(i = 0; i < 3; i++)
65      {
66          NL_OBD_HL_25MS();
67          NL_OBD_SendKLINEFrame(EnterCmd14230, Single, 800, &err);
68          _NL_Delay(1500);
69          if(err == NL_OK)
70          {
71              break;
72          }
73      }
74      return err;
75 }

```

66 行和 67 行两个函数的调用组成了一次初始化过程。66 行进行 fast initialisation 中的把 K 线拉低 25 毫秒和拉高 25 毫秒这个过程。67 行发送一个开始通信的请求数据 (C1 33 F1 81 66)，其服务字节是 0x81，该服务被称为开始通信服务，如果获得汽车 ECU 响应，err 值为 NL_OK。则说明初始化成功。实际测试过程中发现，在确定协议为 ISO14230-4 fast initialisation 情况下，初始化一次就成功的情况并不是一定的。很多年份较久的车型往往要初始化 2 到 3 次初始化才能成功，所以 64 行利用 for 循环在不能成功初始化的情况下会循环 3 次初始化，初始化成功后通过 69 行到 72 行的判断利用 break 语句退出循环，并在 74 行让整个函数返回 NL_OK。下面我们具体看 NL_OBD_HL_25MS 这个函数，代码如下所示。

```

131  */
132  * @描述: K线快速激活函数
133  * @参数: NONE
134  * @返回值: NONE
135  ****
136  void NL_OBD_HL_25MS(void)
137  {
138      _NL_OBD_KLINE_SwPinMode(OUTPUT_PP);
139      _NL_OBD_KLINE_ENABLE();
140      _NL_OBD_KLINE_HEIGHT();
141      _NL_Delay(3000);
142      _NL_OBD_KLINE_LOW();
143      _NL_Delay(25);
144      _NL_OBD_KLINE_HEIGHT();
145      _NL_Delay(25-1);
146      _NL_OBD_KLINE_DISABLE();
147      _NL_OBD_KLINE_SwPinMode(AF_PP);
148      NL_OBD_KLINEConfig(10400);
149 }

```

这个函数完全是按照协议描述进行编写的，在 7.3.1 ISO14230-4 协议解读 中 fast initialisation 描述中使用到协议文档一个截图如下所示。

- first transmission after power on : $T_{idle} \geq W_{5min}$;
- after completion of StopCommunication Service : $T_{idle} \geq P_{3min}$;
- after stopping communication by timeout P_{3max} : $T_{idle} \geq 0ms$.

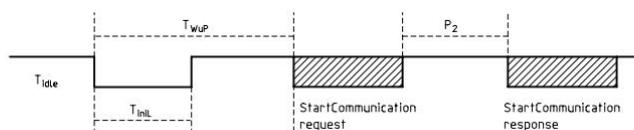


Figure 11 — Fast initialization

Table 10 — Timing values for fast initialization

Parameter	min value, ms	max value, ms
T_{initL}	25 ± 1 ms	24 ms
T_{WuP}	50 ± 1 ms	49 ms

138 行代码把 K 线对应的串口 TX 引脚设置为普通 IO，并设置为推挽式输出。139 行使能 K 线。140 行使 K 线保持高电平。141 行为上图的 Tidle，这里值为 3000 毫秒，实际可以设置为 W5min，也就是 300 毫秒。142 行和 143 行拉低 K 线 25 毫秒对应上图的 TiniL。144 行和 145 行拉高 K 线 25 毫秒对应上图的 TWUP-TiniL。145 行延时函数参数为 25-1，是因为针对本软件利用示波器观察波形时 TWUP 值时 51 毫秒，虽然也能满足上图的定时值的要求，为了精确，25-1 后 TWUP 值为 50 毫秒。146 行失能 K 线，此操作的目的是在进行 147 行恢复 K 线对应串口 TX 复用功能时，不会对 K 线波形产生影响。148 行设置 K 线对应串口波特率为 10400。

再看发送 K 线数据帧函数 NL_OBD_SendKLINEFrame，如下代码截图。

```

92 //*****
93 * @描述: OBD接口发送一帧K线数据
94 * @参数: TxMessage: 待发送K线数据, ReceType接收类型 Timeout: 等待响应时间 , err: 响应是否成功
95 * @返回值: KLINERxRAMDef* 类型指针, 指向响应数据。
96 *****/
97 KLINERxRAMDef* KLINERxRAM;
98 KLINERxRAMDef* NL_OBD_SendKLINEFrame(uint8_t *TxMessage, ReceTypeDef ReceType, uint32_t Timeout, NLStatus *err)
99 {
100     uint8_t acc = 0;
101     uint32_t i;
102     _NL_OBD_KLINE_ENABLE();
103     _NL_OBD_SendKLINEFrame_Init();
104     OBDFlag.RxFlag = NL_FAILURE;
105     for (i = 0; i < TxMessage[0]-1; i++)
106     {
107         acc += TxMessage[i+1];
108     }
109     for (i = 0; i < TxMessage[0]-1; i++)
110     {
111         KLINE_SendByte(TxMessage[i+1]);
112         _NL_Delay(6);
113     }
114     KLINE_SendByte(acc);
115     for (i = 0; i < Timeout; i++)
116     {
117         _NL_Delay(1);
118         if(OBDFlag.RxFlag == NL_SUCCESS && ReceType == Single) break;
119     }
120     if(OBDFlag.RxFlag == NL_SUCCESS)
121     {
122         *err = NL_OK;
123     }
124     else
125     {
126         *err = NL_NOK;
127     }
128     _NL_OBD_KLINE_DISABLE();
129     return &KLINERxRAM;
130 }
```

代码讲解以发送开始通信服务请求数据（Start Communication request）为例，数据如下所示。

```
uint8_t EnterCmd14230[6] = {5, 0xcl, 0x33, 0xf1, 0x81, 0x66};
```

NL_OBD_SendKLINEFrame 函数有四个参数：

参数 1，TxMessage 待发送的数据，数据帧结构必须如上图结构所示，K 线数据帧前增加一个数值表示 K 线数据帧长度。

参数 2，ReceType 待接收数据类型，类型值分别是 Multi 和 Single，表示接收多帧响应或者接收单帧响应。这为法规协议特殊设计，如果是专车整车厂自定义协议不会使用 Multi 类型。

参数 3，Timeout 等待响应数据时间，单位是毫秒。如果汽车 ECU 在此设定值时间内响应 C300 开发板请求，则表示请求成功，否则失败，这将影响参数 4 所指向变量值。

参数 4，err 请求并响应成功与否标志，如果函数发送的请求数据在参数 3 规定时间内获得汽车 ECU 响应，则 err 所指向的存储单元的值是 NL_OK，否则是 NL_NOK。

NL_OBD_SendKLINEFrame 函数返回值是一个指向接收到的响应数据的指针。

函数 102 行和 103 行分别是使能硬件 K 线和初始化与 K 线通信相关的变量，串口设置。104 行设置全局变量 OBDFlag.RxFlag 为 NL_FAILURE，当中断获得汽车 ECU 正确响应该值会变成 NL_SUCCESS。105 行到 108 行计算待发送的请求数据 K 线数据帧的校验字节，所以上图中的开始通信请求数据的最后一个字节 0x66 编写程序的时候并不一定需要给出正确的效验值，

函数会自行计算正确效验值并发送。109 行到 113 行按字节逐个发送 K 线数据帧，字节间隔是 6 毫秒，符合协议中规定的定时参数 P4，它的值要求 5 到 20 毫秒之间。114 行发送计算好的校验字节即当前数据帧 0x66。115 行到 119 行根据 Timeout 参数设置时间等待响应，K 线中断回调函数接收到完整汽车 ECU 响应会使 OBDFlag.RxFlag 值变为 NL_SUCCESS，对应 118 行所示，判断 OBDFlag.RxFlag 值为 NL_SUCCESS 并且接收类型 ReceType 为 Single 将立刻退出循环。120 行到 127 行判断 OBDFlag.RxFlag 值为 NL_SUCCESS 可断定响应数据是否接收成功，成功给 err 所指向的存储单元赋值 NL_OK，失败赋值 NL_NOK。那么如果接收类型 ReceType 是 Multi 时，115 行到 119 行将不会主动退出循环，直到等待时间达到 Timeout 参数设置时间为止。128 行失能 K 线，129 行返回 KLINERxRAM 二维数组地址，该数组存储汽车 ECU 响应的数据，存储是通过 K 线中断回调函数完成。

下面是 K 线回调函数代码截图。

```

579 /*************************************************************************/
580 * @描述: KLINE接收回调函数
581 * @参数:  NONE
582 * @返回值: NONE
583 /*************************************************************************/
584 uint8_t KFrameLen[20] = {0};
585 uint8_t KFrameLenCount = 0;
586 void KLINERxReceiveCallBack(void)
587 {
588     uint8_t i, j;
589     if(KLINELen(RXBUFFERSIZE-huart1.RxXferCount-KFrameLen[KFrameLenCount], &KLINERxBuffer[KFrameLen[KFrameLenCount]]) == NL_OK)
590     {
591         KFrameLen[+KFrameLenCount] = RXBUFFERSIZE-huart1.RxXferCount;
592     }
593     if(KFrameLenCount >= 2)
594     {
595         KLINERxRAM.Count = KFrameLenCount-1;
596         for(j = 0; j < KFrameLenCount; j++)
597         {
598             for(i = 0; i < KFrameLen[2+j]; i++)
599             {
600                 KLINERxRAM.RxRAM[j][i] = KLINERxBuffer[KFrameLen[1+j]+i];
601             }
602         }
603         if(KLINERxRAM.RxRAM[0][0]!=0)
604         {
605             OBDFlag.RxFlag = NL_SUCCESS;
606         }
607     }
608     if(KLINERxBuffer[0] == 0x55 && RXBUFFERSIZE-huart1.RxXferCount == 3)
609     {
610         for(i = 0; i < 3; i++)
611         {
612             KLINERxRAM.RxRAM[0][i] = KLINERxBuffer[i];
613         }
614         OBDFlag.RxFlag = NL_SUCCESS;
615     }
616     if(KLINERxBuffer[0] == 0x55 && RXBUFFERSIZE-huart1.RxXferCount == 5)
617     {
618         for(i = 0; i < 5; i++)
619         {
620             KLINERxRAM.RxRAM[0][i] = KLINERxBuffer[i];
621         }
622         OBDFlag.RxFlag = NL_SUCCESS;
623     }
624 }
625 }
```

进入该回调函数前，串口接收到的 K 线数据已经被存储于 KLINERxBuffer 内存区域中，它由下面调用 HAL 库函数自动完成。

```

/* ************************************************************************
* @描述: 串口回调中断初始化函数
* @参数: UART_HandleTypeDef *UartHandle
* @返回值: void
************************************************************************/
void USER_UART_RxInitCallback(UART_HandleTypeDef *huart)
{
    huart->RxState = HAL_UART_STATE_READY;
    if(huart->Instance == USART1)
    {
        HAL_UART_Receive_IT(huart, (uint8_t *)KLINERxBuffer, RXBUFFERSIZE);
    }
    else if(huart->Instance == USART2)
    {
        HAL_UART_Receive_IT(huart, (uint8_t *)IOTRxBuffer, RXBUFFERSIZE);
    }
    else if(huart->Instance == USART3)
    {
        HAL_UART_Receive_IT(huart, (uint8_t *)GPSRxBuffer, GPSRXBUFFERSIZE);
    }
}
```

KLINERxBuffer 用于存储接收的数据，RXBUFFERSIZE 值表示接收数据的长度，单位为字节。了解这个对 K 线接收回调函数的理解就容易了。

589 行判断接收的数据是否满足 K 线数据帧结构，主要是通过 KLINELen 函数计算校验字节是否满足条件判断，其函数原型如下所示。

```

/* @描述: ACC效验
 * @参数: LEN 数据长度 ram要处理的内存指针
 * @返回值: NL_OK 效验成功 NL_NOK 效验失败
 */
NLStatus KLINELen(uint8_t len,uint8_t *ram)
{
    uint8_t i,acc = 0;
    for(i = 0; i < len-1; i++)
    {
        acc += ram[i];
    }
    if(ram[len-1] == acc && len != 1)
    {
        return NL_OK;
    }
    return NL_NOK;
}

```

参数 1, len 表示待计算内存的长度。参数 2, ram 表示待计算内存的起始地址。如果返回值是 NL_OK 表示校验成功，符合 K 线数据帧结构，否则不符合。

591 行把符合 K 线数据帧的长度存储于数组 KFrameLen 中。503 到 608 行判断接收的 K 线数据帧大于等于 2 的情况下，开始存储接收到的 K 线数据帧。之所以要大于等于 2 帧才接收是因为 K 线是收发共用的物理通信线，收到的第一帧为请求数据，第二帧才是汽车 ECU 响应数据。595 行把响应的帧数量存储于 KLINERxRAM.Count 中，方便用户得知存储了多少帧响应数据。596 行到 602 行开始存储响应数据于 KLINERxRAM.RxRAM 数组中。603 行到 606 行简单判断 KLINERxRAM.RxRAM 存储了内容后即可给 OBDFlag.RxFlag 变量赋值 NL_SUCCESS，告知主程序接收响应数据完成。609 行到 616 行专用于接收初始化函数中接收 0x55 同步参数，KB1 和 KB2 存储于数组 KLINERxRAM.RxRAM[0][0], KLINERxRAM.RxRAM[0][1] 和 KLINERxRAM.RxRAM[0][2] 中，接收完毕给 OBDFlag.RxFlag 变量赋值 NL_SUCCESS 告知主程序。617 行到 624 行存储接收到的同步参数，KB1, KB2, 取反 KB2 和取反地址信息于 KLINERxRAM.RxRAM[0][0], KLINERxRAM.RxRAM[0][1], KLINERxRAM.RxRAM[0][2], KLINERxRAM.RxRAM[0][3], KLINERxRAM.RxRAM[0][4]，接收完毕给 OBDFlag.RxFlag 变量赋值 NL_SUCCESS 告知主程序。

ISO14230-4 协议应用层代码的实现

1. 读车架号

在 NL_OBD_SendKLINEFrame 函数的支持下，应用层和描述层的代码之下的协议已由该函数实现。至此我们只需要关注应用层功能开发即可。

首先看读车架号功能在 C300 开发板中是如何实现的，下面是读车架号的应用层代码，无论快速初始化还是 5 波特率初始化，功能代码都是完全一致的。

```

198     if(OBDStruct.VINStruct.flag == RESET)
199     {
200         NL_ClearRAM((uint8_t*)OBDStruct.VINStruct.VIN,18);
201         ram = ISO14230_4_GetVIN(&err);
202         if(err == NL_OK)
203         {
204             strncpy(OBDStruct.VINStruct.VIN,(const char*)ram,17);
205             OBDStruct.VINStruct.flag = SET;
206         }
207     }

```

该代码结构和其它协议的读车架号功能应用层代码是保持一致的。198 行

OBDStruct.VINStruct.flag 变量用于判断是否需要读取车架号，值为 RESET 时需要读取，值为 SET 表示不需要读取，车架号信息待 TCPTASK 任务上传服务器。其实即使 TCPTASK 任务上传服务器后该变量仍然保持为 SET。读车架号只存在于设备上电识别协议后读取一次车架号，只要是 C300 车联网开发板不重新上电，该变量会一直为 SET，只允许读取一次车架号信息。因为实际车辆车架号出厂后一直固定不变的，不会发生随机改变。使用我们的模拟器模拟车架号信息给 C300 开发板读取时，请在协议选择并设置好车架号后再接入 C300 开发板，尽可能模拟实车环境。200 行对存储车架号信息的 OBDStruct.VINStruct.VIN 变量清空待车架号存入。201 行是获取车架号函数，参数是传递引用参数，如果读取车架号失败，该参数所指向的存储单元只为 NL_NOK，读取车架号成功该值为 NL_OK，返回值便是车架号存储位置的内存地址。203 行做对 err 参数进行判断，读取车架号成功后 204 行把车架号信息存储在 OBDStruct.VINStruct.VIN 中供系统其它任务进行调用。204 行在车架号读取成功并存储后对 OBDStruct.VINStruct.flag 赋值 SET。提示 TCPTASK 任务可随时调用上传服务器。下面再看 ISO14230_4_GetVIN 函数原型，代码如下所示。

```

76 //*****
77 * @描述: ISO14230-4获取车架号
78 * @参数: NONE
79 * @返回值: 车架号存储指针
80 *****/
81 char* ISO14230_4_GetVIN(NLStatus *err)
82 {
83     uint8_t i,j;
84     KLINERxRAMDef *ram;
85     ram = NL_OBD_SendKLINEFrame(VINCmd14230,Multi,800,err);
86     _NL_Delay(150);
87     if(*err == NL_OK)
88     {
89         for(j = 0; j < ram->Count; j++)
90         {
91             for(i = 0; i < 4; i++)
92             {
93                 VINRAM[i+j*4] = ram->RxRAM[j][6+i];
94             }
95         }
96     }
97     VINRAM[17] = 0;
98     return VINRAM;
99 }

```

刚才已经对该函数的参数和返回值进行了介绍，直接看函数代码 85 行调用函数 NL_OBD_SendKLINEFrame，参数分别是车架号 ISO14230-4 协议 K 线请求数据，接收类型为 Multi 多帧，等待汽车 ECU 响应时间是 800 毫秒。这时候我们可以预见，汽车 ECU 对车架号的响应数据必须为多帧响应，因为前面协议解读我们知道，ISO14230-4 协议每帧 K 线数据帧只允许承载 7 个字节的应用层数据，除去服务字节，信息类型，信息计数，对于车架号响应信息而言，每帧 K 线数据帧只承载车技号信息只有 4 个字节，如果 NL_OBD_SendKLINEFrame 函数最后一个传递引用参数所指向的变量值为 NL_OK，表示车架号响应成功，函数返回值的指针所指向地址将完整存储包括服务字节，信息类型，信息计数和车架号信息。所以 89 行到 95 行把车架号以外的信息去除，只存储车架号信息于数组 VINRAM 中。97 行在 17 位车架号信息后加上结束字符 0。98 行返回存储车架号信息的数组 VINRAM 地址。

2. 读故障码

读故障码的代码片段如下截图所示。

```

208 if(ISO14230_4_GetNotDrivingState() == NL_OK && OBDStruct.DTCStruct.flag == RESET)
209 {
210     ram = ISO14230_4_GetDTC(&err);
211     if(err == NL_OK)
212     {
213         NL_ClearRAM((uint8_t*)OBDStruct.DTCStruct.DTC,100);
214         strcpy(OBDStruct.DTCStruct.DTC,ram);
215         OBDStruct.DTCStruct.flag = SET;
216     }
217 }

```

208 行有两个判断条件满足情况下才可以读故障码，ISO14230_4_GetNotDrivingState() 用于判断当前车速是否为 0，函数返回值为 NL_OK 表示当前车速为 0。C300 开发板默认程序要求车速为 0 的情况下才可以读故障码信息。并且要求 OBDStruct.DTCStruct.flag 为 RESET 才允许读取故障码信息。210 行调用 ISO14230_4_GetDTC 函数获取故障码信息，函数返回值为已经解析编码后的故障码，传递引用参数 err 的值为 NL_OK 表示故障码获取成功。所以 211 行到 216 行，把 ISO14230_4_GetDTC 函数返回值即故障码号拷贝到全局变量 OBDStruct.DTCStruct.DTC 中。并设置 OBDStruct.DTCStruct.flag 变量值为 SET。下面我们具体看 ISO14230_4_GetDTC 函数代码，了解它的实现过程。

```

101 /* ****
102 * @描述: ISO14230读取故障码
103 * @参数: *err :NL_OK:成功 NL_NOK:不成功
104 * @返回值: 故障码存储指针
105 ****/
106 char* ISO14230_4_GetDTC(NLStatus *err)
107 {
108     uint8_t i,j;
109     KLINERxRAMDef *ram;
110     uint16_t dtc;
111     NL_ClearRAM((uint8_t*)DTCRAM,200);
112     ram = NL_OBD_SendKLINEFrame(DTCCmd14230,Multi,800,err);
113     _NL_Delay(150);
114     if(*err == NL_OK)
115     {
116         for(j = 0; j < ram->Count; j++)
117         {
118             for(i = 0; i < 3; i++)
119             {
120                 if(ram->RxRAM[j][4+2*i] != 0 && ram->RxRAM[j][5+2*i] != 0)
121                 {
122                     dtc = ram->RxRAM[j][4+2*i]<<8|ram->RxRAM[j][5+2*i];
123                     strncpy(DTCRAM+strlen(DTCRAM),(const char*)NL_PCBU(dtc),5);
124                     strcpy(DTCRAM+strlen(DTCRAM),":");
125                 }
126             else
127             {
128                 DTCRAM[strlen(DTCRAM)-1] = 0;
129                 return DTCRAM;
130             }
131         }
132     }
133     DTCRAM[strlen(DTCRAM)-1] = 0;
134     return DTCRAM;
135 }
136 }
```

111 行清空 DTCRAM 数组，准备存储故障码号。112 行调用 NL_OBD_SendKLINEFrame 函数，通过参数 DTCCmd14230 当前故障码请求数据，等待故障码多帧响应，如果故障码超过 3 个将使用多帧接收类型才能接收完整数据，所以这里不管是少于 3 个故障码还是多于 3 个故障码将统一采用多帧接收类型等待故障码完整响应。114 行到 133 行对 NL_OBD_SendKLINEFrame 获得的故障码响应数据处理，去除服务字节，并通过 123 行的 NL_PCBU 函数对十六进制的故障码原始数据转换成以 ASCII 编码的文本故障码，并存储在 DTCRAM 数组中。134 行在存储故障码结果后的 DTCRAM 数组再其存储的故障码结尾加上结束字符 0。135 行返回 DTCRAM 数组地址。

3. 读数据流

下面是 ISO14230-4 协议读数据流的代码片段。

```

218 if(OBDStruct.DSStruct.flag == RESET)
219 {
220     dsram = ISO14230_4_GetDS(ISODSItem,&err);
221     if(err == NL_OK)
222     {
223         OBDStruct.LINKSTATUS = SET;
224         OBDStruct.DSStruct.Total = dsram->Total;
225         for(i = 0; i < OBDStruct.DSStruct.Total; i++)
226         {
227             strncpy(OBDStruct.DSStruct.DS[i],dsram->DS[i],30);
228         }
229         OBDStruct.DSStruct.flag = SET;
230     }
231     else
232     {
233         OBDStruct.LINKSTATUS = RESET;
234         NL_LED_ONOFF(LEDOBD,OFF,Fashing,LED1HZ);
235         break;
236     }
237 }

```

218 行用于判断是否读数据流，如果 OBDStruct.DSStruct.flag 值为 RESET，系统要求读取数据流，如果是 SET 表示数据流读取完毕，但未被上传服务器。220 行读取数据流函数，函数的参数有两个，参数 1，ISODSItem 表示要读取数据流的项目，参数 2，传递引用参数 err 用于判断数据流读取是否成功，函数返回值是读到的数据流结果，以 ASCII 格式编码。如果数据流读取成功执行 221 行到 230 行，其中 223 行的 OBDStruct.LINKSTATUS 值为 SET 表示 C300 开发板与汽车 ECU 处于链接状态。224 行结构体 OBDStruct 用于存储待上传的 OBD 诊断数据，OBDStruct.DSStruct 用于存储待上传的数据流，OBDStruct.DSStruct.Total 用于存储读取数据流的个数。225 行到 228 行分别把读到的数据流存储到 OBDStruct.DSStruct.DS 数组中。229 行数据流读取完毕，把 OBDStruct.DSStruct.flag 值设置为 SET。如果 ISO14230_4_GetDS 函数读取数据流失败，err 值为 NL_NOK，执行 232 行到 236 行代码，设置 OBDStruct.LINKSTATUS 值为 RESET，告知系统 C300 开发板与汽车 ECU 通信失败，TCP 任务会做出响应动作，告知服务器，同时 234 行关闭 OBD 灯。235 行通过 break 退出 ISO14230 协议通信。接下来具体看 ISO14230_4_GetDS 函数代码的实现，代码截图如下所示。

```

165 //*****ISO14230读取数据流*****
166 //参数: uint8_t *item: 数据流项目索引 ,*err :NL_OK:成功 NL_NOK:不成功
167 //返回值: 数据流存储指针
168 //*****DSStructDef* ISO14230_4_GetDS(DSItemStructDef item,NLStatus *err)
169 DSStructDef* ISO14230_4_GetDS(DSItemStructDef item,NLStatus *err)
170 {
171     uint8_t i;
172     KLINERxRAMDef *ram;
173     DSStruct.Total = item.Total;
174     for(i = 0; i < item.Total; i++)
175     {
176         DS Cmd14230[5] = DSCmd15301[item.Item[i]].PIDByte;
177         ram = NL_OBD_SendKLINEFrame(DSCmd14230,Single,800,err);
178         NL_Delay(150);
179         if(*err == NL_OK)
180         {
181             ErrorCount = 0;
182             if (DSCmd15301[item.Item[i]].Type == Numeric)
183             {
184                 sprintf(DSStruct.DS[i],DSCmd15301[item.Item[i]].Format,DSCmd15301[item.Item[i]].Equation0(ram->RxRAM[0]+DSCmd15301[item.Item[i]].FineByte+2));
185             }
186             else
187             {
188                 strcpy(DSStruct.DS[i],"");
189                 strcpy(DSStruct.DS[i],DSCmd15301[item.Item[i]].Equation1(ram->RxRAM[0]+DSCmd15301[item.Item[i]].FineByte+2));
190                 strcpy(DSStruct.DS[i]+strlen(DSStruct.DS[i]),"\n");
191             }
192         }
193         else
194         {
195             if (++ErrorCount > 4)
196             {
197                 *err = NL_NOK;
198                 return &DSStruct;
199             }
200         }
201     }
202 }
203
204 return &DSStruct;
205 }

```

函数读到的数据流结果将被暂时存储在 DSStruct 中。173 行把参数 1 的数据流个数存储在 DSStruct.Total 中，参数 1 的参数如下图所示。

```
DSItemStructDef ISODSItem = {6,48,49,40,41,47,51};
```

它分别由数据流个数和数据流索引值组成。174 行到 202 行是上图 6 个数据流读取的过程。176 行给 DSCmd14230 数据流请求数据根据数据流索引替换对应 PID 值，替换的位置如下红色方框所示。

```
/******读数据流*****/uint8_t DSCmd14230[7] = {6,0xC2,0x33,0xF1,0x01,0x00,0x00};
```

PID 等相关信息存储在结构体数组变量 DSControl15301 中，其中 DSControl15301[].PIDByte 成员存储的就是 PID 值。DSControl15301 结构体数组如下所示，截图源自 ISO15031_5.C 文件。

```
/*
*****数据流控制*****
const DSControl15301TypeDef DSControl15301[DSTotal1X] = {
    {Numeric ,0x01,3,"%f",Formula000,NONE },//DS000 ECU中存储的故障码数量
    {Character,0x01,3,"",NONE ,Formula001},//DS001 MIL(故障指示灯)状态
    {Character,0x01,4,"",NONE ,Formula002},//DS002 支持失火监测
    {Character,0x01,4,"",NONE ,Formula003},//DS003 支持燃油系统监测
    {Character,0x01,4,"",NONE ,Formula004},//DS004 支持综合部件监测
    {Character,0x01,4,"",NONE ,Formula005},//DS005 失火监测准备就绪
    {Character,0x01,4,"",NONE ,Formula006},//DS006 燃油系统监测准备就绪
    {Character,0x01,4,"",NONE ,Formula007},//DS007 综合部件监测准备就绪
    {Character,0x01,5,"",NONE ,Formula008},//DS008 支持NMHC催化剂监测(清码后)
    {Character,0x01,5,"",NONE ,Formula009},//DS009 支持氮氯化合物后处理监测(清码后)
    {Character,0x01,5,"",NONE ,Formula010},//DS010 支持增压压力系统监测(清码后)
    {Character,0x01,5,"",NONE ,Formula011},//DS011 支持废气传感器监测(清码后)
    {Character,0x01,5,"",NONE ,Formula012},//DS012 支持PM(颗粒物)过滤器监测(清码后)
    {Character,0x01,5,"",NONE ,Formula013},//DS013 支持EGR(废气再循环)系统和/or VVT(可变阀门正时)系统监测(清码后)
    {Character,0x01,6,"",NONE ,Formula014},//DS014 NMHC催化剂监测准备就绪(清码后)
    {Character,0x01,6,"",NONE ,Formula015},//DS015 氮氯化合物后处理监测准备就绪(清码后)
    {Character,0x01,6,"",NONE ,Formula016},//DS016 增压压力系统监测准备就绪(清码后)
    {Character,0x01,6,"",NONE ,Formula017},//DS017 废气传感器监测准备就绪(清码后)
    {Character,0x01,6,"",NONE ,Formula018},//DS018 PM(颗粒物)监测准备就绪(清码后)
    {Character,0x01,6,"",NONE ,Formula019},//DS019 支持EGR(废气再循环)系统和/or VVT(可变阀门正时)系统监测准备就绪(清码后)
    {Character,0x41,4,"",NONE ,Formula020},//DS020 失火检测激活
    {Character,0x41,4,"",NONE ,Formula021},//DS021 燃油系统检测激活
    {Character,0x41,4,"",NONE ,Formula022},//DS022 综合部件检测启用
    {Character,0x41,4,"",NONE ,Formula023},//DS023 失火检测完成
    {Character,0x41,4,"",NONE ,Formula024},//DS024 燃油系统检测完成
    {Character,0x41,5,"",NONE ,Formula025} //支持NMHC催化剂监测(驾驶循环)
};
```

第一列成员表示数据类型，其值有 Numeric (数值型) 和 Character (文本型)。第二列成员表示 PID 值。第三列成员表示取值位置，当前协议下 3, 4, 5, 6 分别对应 ISO15031-5 协议所描述的 A 字节，B 字节，C 字节，D 字节。第四列成员用于输出数值格式控制，等同于 printf 函数中的格式控制字符，上图所示的 DS000，“%.0f” 表示输出的数值为浮点型，但是不保留小数，也就等同于整数显示。第五列成员，当数据流为 Numeric (数值型) 时，调用该列函数进行分辨率转换。第六列成员当数据流为 Character (文本型) 调用该列函数进行分辨率转换。

了解 DSControl15301 结构体数组后，我们继续看代码 177 行，调用函数 NL_OBD_SendKLINEFrame 请求对应数据流索引的数据流，这时候函数的接收类型设置为 Single，ISO14230-4 协议数据流都是一帧请求数据，对应一帧响应数据流。如果 NL_OBD_SendKLINEFrame 函数获得汽车 ECU 正确响应，执行 180 行到 193 行代码。如果没有获得汽车 ECU 响应执行 195 行到 201 行代码，其中 196 行 ErrorCode 变量会加 1，该变量用于计数获得失败响应的数据流个数，如果失败响应超过 5 次，198 行通过给 err 赋值 NL_NOK 判定与汽车 ECU 失去通信。下面我们回头看获得正确响应的代码，181 行在获得汽车 ECU 正确响应的情况下给 ErrorCode 变量清零。182 行到 186 行，如果数据流的数据类型为 Numeric (数值型)，通过 184 行，将取到的响应数据根据 DSControl15301 结构体的描述进行分辨率转换获得最终结果存储在 DSStruct.DS 数组中。该语句看起来有点复杂，但是我们拆分来看就比较容易明白，sprintf 函数是 C 语言标准库函数，与 printf 函数类似，但是不同之处在于 printf 函数把内容打印到屏幕上，而 sprintf 函数是打印内容到指定内存。sprintf 函数有 3 个参数，参数 1，打印内容到所存储的地址，参数 2，格式控制，参数 3 待打印的内容。所以参数 1 是 DSStruct.DS 数组，前面已经说过，S014230_4_GetDS 函数执行的结果被暂时存储在 DSStruct 结构体中。参数 2，DSControl15301[item.Item[i]].Format 格式控制，其值是 DSControl15301 结构体数组对应数据流索引第 4 列内容。item.Item[i] 是当前执行的数据流索引。参数 3，

DSControl15301[item.Item[i]].Equation0(ram->RxRAM[0]+DSControl15301[item.Item[i]].FineByte+2)，这个参数看起来有点长，但是继续把它拆分两部分就很容易理解了，第一部分：DSControl15301[item.Item[i]].Equation0()，调用的是 DSControl15301 结构体数组对应索引 item.Item[i] 的第 5 列成员，该成员是分辨率转换函数；第二部分：ram->RxRAM[0]+DSControl15301[item.Item[i]].FineByte+2 根据 DSControl15301 结构体

数组对应索引 item.Item[i] 的第三列成员找到响应数据对应取值，该取值作为 DSControl15301[item.Item[i]].Equation0() 参数。 DSControl15301[item.Item[i]].Equation0 为函数指针。以上图的 DS000 ECU 存储的故障码数量为例，响应数据的取值调用 Formula000 进行分辨率转换。下面截图是 Formula000 函数的源码。

```
/* **** */
 * @描述: ECU中存储的故障码数量
 * @参数: u8 *data
 * @返回值: float
 ****/
float Formula000(u8 *p)
{
    return (*p & 0x7F);
}
```

所以 DS000 ECU 存储的故障码数量这个数据流就是取响应数据的 A 字节代入 Formula000 函数计算，函数取 A 字节的低 7 位表示故障码数量，并按照格式控制 "%.0f" 输出到数组 DSStruct.DS 中。

如果数据流是 Character (文本型)，执行 189 行到 191 行代码。189 行和 191 行两行代码只是给转换后的值加上了单引号。重点看 190 行代码，190 行代码也使用了 C 语言标准库函数 strcpy，顾名思义，就是字符串拷贝函数，参数有两个，功能就是把参数 2 的内容追加拷贝到参数 1。参数 2 为

DSControl15301[item.Item[i]].Equation1(ram->RxRAM[0]+DSControl15301[item.Item[i]].FineByte+2)。函数指针 DSControl15301[item.Item[i]].Equation1 调用的是 DSControl15301 结构体数组第 6 列的成员，参数是 ram->RxRAM[0]+DSControl15301[item.Item[i]].FineByte+2 根据 DSControl15301 结构体数组对应索引 item.Item[i] 的第三列成员找到响应数据对应取值。以 DS001 MIL(故障灯) 状态为例，调用 Formula001 函数进行分辨率转换，如下图所示。

```
/* **** */
 * @描述: MIL(故障指示灯)状态
 * @参数: u8 *p
 * @返回值: float
 ****/
char* Formula001(u8 *p)
{
    if ((*p&0x80) != 0)
    {
        return "ON";
    }
    return "OFF";
}
```

取响应数据的 A 字节的第 7 位进行判断，如果是 1 返回文本 ON，如果是 0 返回文本 OFF。最后把该返回值通过 strcpy 存储到 DSStruct.DS 数组中。

7.3.3 ISO14230-4 协议以及在开发板中的程序实现视频教程

7.3.1 ISO14230-4 协议解读视频教程

视频教程链接: <https://www.bilibili.com/video/BV1iE411H7dK?p=14>

7.3.2 ISO14230-4 协议在 Neulen TBOX C300 开发板中的程序实现

视频教程链接: <https://www.bilibili.com/video/BV1iE411H7dK?p=15>

7.4 ISO9141-2 协议以及在开发板中的程序实现

7.4.1 ISO9141-2 协议解读

ISO9141-2 协议相对于前面几个协议在当前的实车中应用相对较少, 主要集中在早期版本的大众和三菱车型, 如 2010 年及以前的捷达和桑塔纳等车型, 三菱 4G 发动机, 以及国产车型中仿制三菱 4G 发动机早期版本, 因为这些仿制的国产发动机大部分继续采用仿制前的发动机电控系统。我们对 ISO9141-2 协议学习相对容易, 学习内容主要解读 ISO9141-2 这份协议内容即可, 同时有前面几份协议的解读经验就更容易找到我们需要的协议信息。

1. 协议通信如何唤醒?

ISO9141-2 协议物理层采用 K 线进行通信, 唤醒即初始化方式采用 5 波特率方式进行初始化, 初始化过程如下图所示 (本节协议截图均源自 ISO9141-2 协议)。

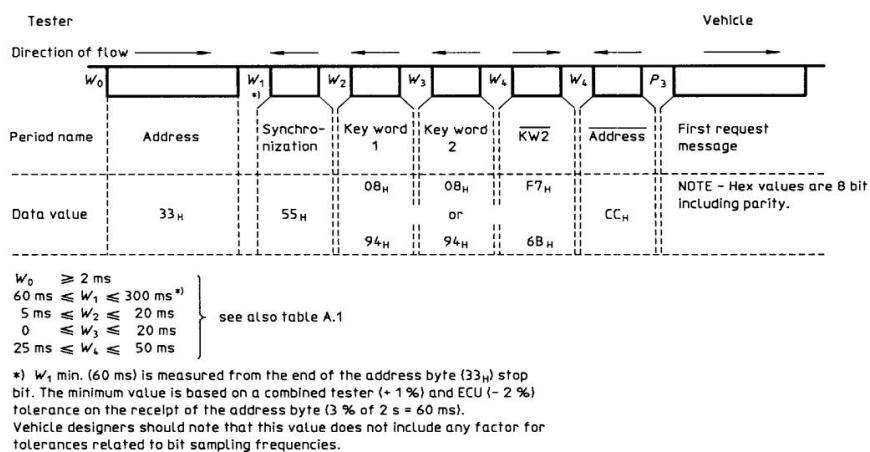


Figure 6 — Header

初始化过程和 ISO14230-4 协议的 5 波特率初始化是一样的。测试设备以 5 波特率发送地址信息 $0x33$, 等待 60 毫秒到 300 毫秒汽车 ECU 响应同步信息 $0x55$, keyword1 和 keyword2, 测试设备接收到这些响应数据之后, 在 25 毫秒到 50 毫秒之间发送 keyword2 取反值给汽车 ECU, 汽车 ECU 接收到该字节后也是在 25 毫秒到 50 毫秒之间响应地址信息的取反值, 测试设备接收到这个字节之后, 在定时参数 P_3 时间内, 也就是 55 毫秒到 5 秒钟内向汽车 ECU 发送第一个请求信息, 如果请求信息获得了汽车 ECU 的响应, 则证明初始化成功。所以在 C300 开发板中 ISO14230-4 协议和 ISO9141-2 协议的 5 波特率初始化可以共用函数 NL_OBD_Wakeup_Addr。但是值得注意的一点, ISO9141-2 协议的 Keyword 值只有两个版本分别是 $0x08$, $0x08$ 和 $0x94$, $0x94$ 。这两个版本只对定时参数设置产生作用, 具体内容在本节的第 3 部分通信信息管理再具体介绍。

2. 数据格式

ISO9141-2 协议数据格式如下图所示。

Table 1 — Message structure

Sequence	Description	Column 1	Column 2	Reference
Byte 1	Message header 1	68 _H	48 _H	SAE J1979
Byte 2	Message header 2	6A _H	6B _H	SAE J1979
Byte 3	Source address	F1 _H	Address	SAE J1979
Bytes 4 to 10	Data (up to 7 bytes)	XX _H	XX _H	SAE J1979
Final byte	Checksum	YY _H	YY _H	see 11.2

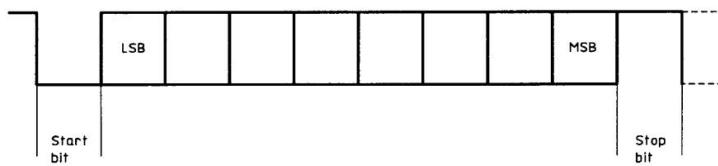


Figure 7 — Byte format

如 Table1 所示，数据帧结构由 Message header1 + Message header2 + Source address + data(应用层原始数据)+Checksum。区分请求还是响应数据可以通过 Message header1 和 Message header2 进行辨别。当 Message header1 和 Message header2 分别是 0x68 和 0x6A 时，该数据帧表示请求数据，如 Table1 中的 Column1 所示。当 Message header1 和 Message header2 分别是 0x48 和 0x6B 时，该数据帧表示响应数据，如 Table1 中的 Column2 所示。

第三字节表示源地址，对于请求数据来说，源地址就是诊断服务设备所以源地址固定为 0xF1。对于响应数据来说，源地址是作出响应的 ECU 地址，如果是 ECU#1 的话是 0x11。第四字节开始承载应用层原始数据，承载的字节数量可以是 1 个字节到 7 个字节。数据帧格式最后一个字节是校验字节，和 ISO14230-4 协议的校验方法一样是累加和校验，就是把数据帧每个字节进行累加取最低字节作为校验结果。Table1 中数据帧结构协议要求参考 SAEJ1979，其实就是 ISO15031-5 协议，在 SAE 中，ISO15031-5 协议被命名为 SAEJ1979。Figure7 展示了单个字节是如何传输的，由起始位，8 位数据位，和一个结束位组成，字节从低位开始传输，可见就是串口的传输模式。

3. 通信信息管理

这部分通信内容基本和 ISO14230-4 协议类似，ISO9141-2 协议依然只能一帧数据承载 7 个字节的应用层数据，所以遇到响应多个故障码信息或者车架号信息的时候只能通过多帧响应完成信息的传输。这将在后面应用层协议中具体展示。通信信息管理还有一项关键内容就是定时参数设置。前面通信协议唤醒内容提及 ISO9141-2 协议由两个版本的 Keyword，分别是 0x08, 0x08 和 0x94, 0x94，它会对定时参数设置会产生影响，如下图所示。

Table A.2 — Minimum and maximum values of protocol timing

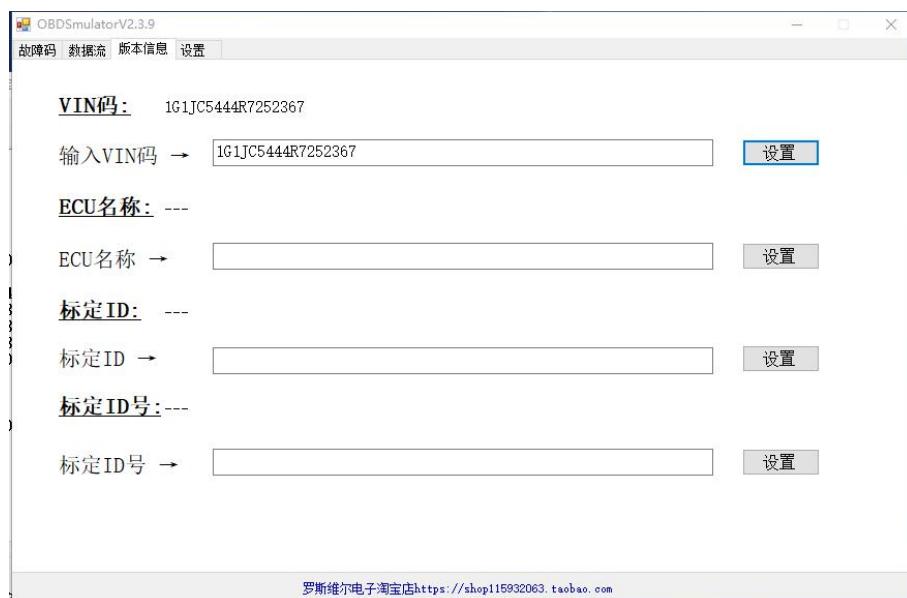
Symbol	Time		Description
	min.	max.	
P_1	0	20 ms	Interbyte time for messages from the vehicle to the diagnostic tester (see clause 12)
P_2 (94)	0	50 ms	Intermessage time for vehicles with key word of 94H (see clauses 12 and 13)
P_2 (08)	25 ms	50 ms	Intermessage time for vehicles with key word of 08H (see clauses 12 and 13)
P_3	55 ms	5 s	Intermessage time between the end of all vehicle-sourced responses and the start of the next diagnostic tester request (see clauses 12 and 13)
P_4	5 ms	20 ms	Interbyte time for messages from the diagnostic tester to the vehicle (see clauses 12 and 13)

产生影响的定时参数设置主要是 P2, P2 表示汽车 ECU 接收到请求信息做出响应的时间，如果 Keyword 是 0x94, 0x94 的话，接收到请求并做出响应时间会在 0 到 50 毫秒时间内。如果 Keyword 是 0x08, 0x08 的话，接收到请求数据并做出响应时间在 25 毫秒到 50 毫秒时间内。但是这个细微的变化对我们程序的编写似乎作用不大，我们只需设置等待响应时间在 50 毫秒及以上即可，所以我们完全可以不关注 Keyword 的值。另外 P1 表示汽车 ECU 响应数据字节间时间间隔，要求 20 毫秒以内。P3 表示汽车 ECU 响应结束到诊断设备新的请求时间，在 55 毫秒到 5 秒时间内，如果超过 5 秒没有新的请求信息，汽车 ECU 将进入诊断通信休眠，需要再次通信就必须再进行 5 波特率初始化。P4 表示请求数据字节间时间间隔，5 毫秒到 20 毫秒间。

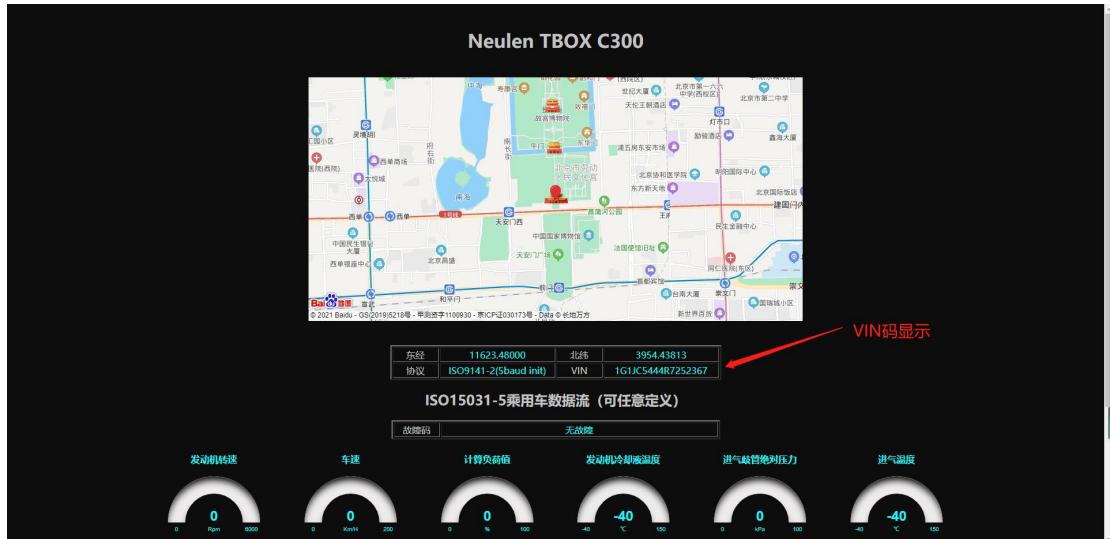
4. 应用层定义

应用层定义与前面的 ISO14230-4 协议的定义完全一致，定义于 ISO15031-5 协议，所以这里不再详细讲述 ISO15031-5 协议文档。我们直接采集数据对照进行讲述。

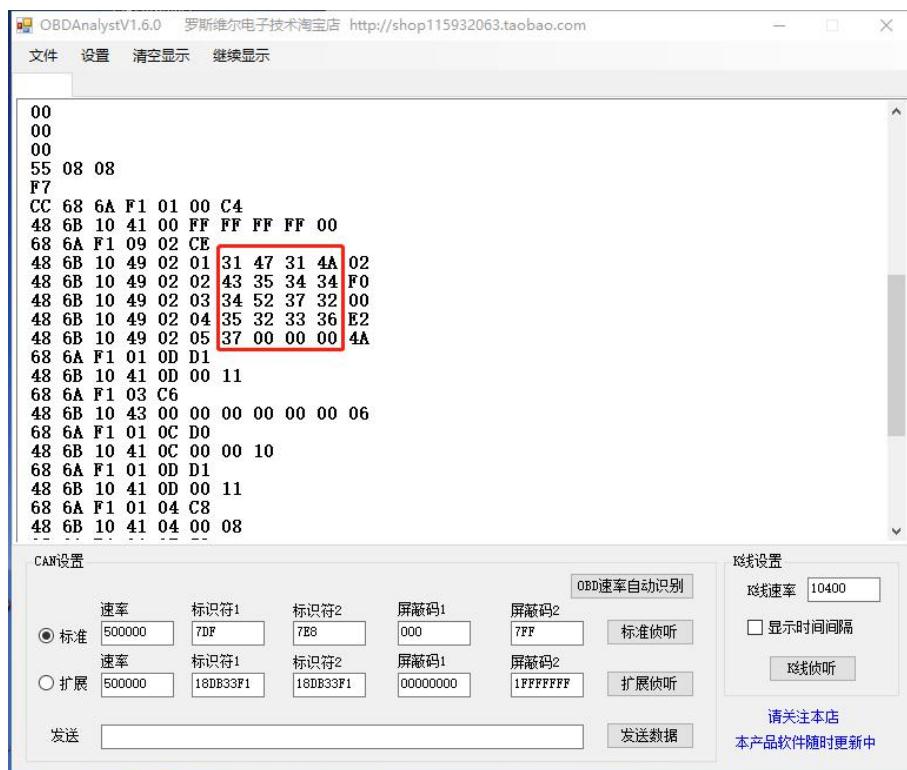
模拟器在选择 ISO9141-2 协议后，如下图所示设置模拟器 VIN 码为 1G1JC5444R7252367。



C300 开发板读到结果如下图所示。



读取车架号过程可以通过分析仪采集数据如下所示。



图中红色方框内的数据就是车架号数据，以 ASCII 编码。和 ISO14230-4 协议一样，响应的每帧数据只能承载 7 个字节的应用层数据，而承载车架号信息每帧只能承载 4 个字节，所以需要多帧进行响应。

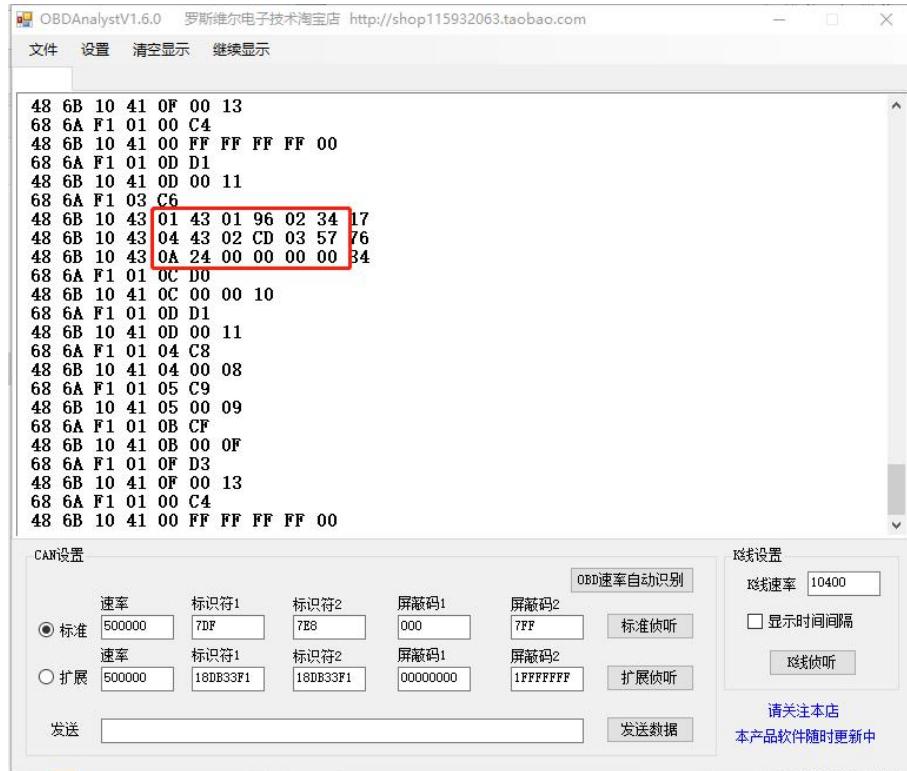
读故障码过程，也可以进行如下设置，首先设置模拟器模拟故障码。此处，我们设置 7 个故障码分别是 P0143, P0196, P0234, P0443, P02CD, P0357, P0A24。



C300 开发板读到的故障码如下图所示。

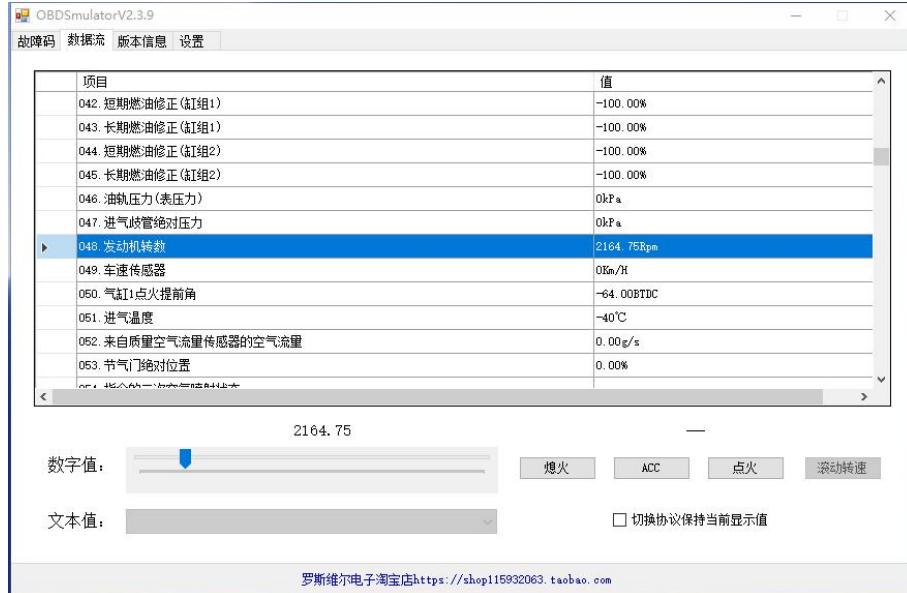


读取故障码过程可以通过分析仪采集数据如下所示。



红色方框内的数据就是模拟的 7 个故障码数据，2 个字节表示一个故障码，一帧响应数据只能承载 3 个故障码，需多帧进行响应。

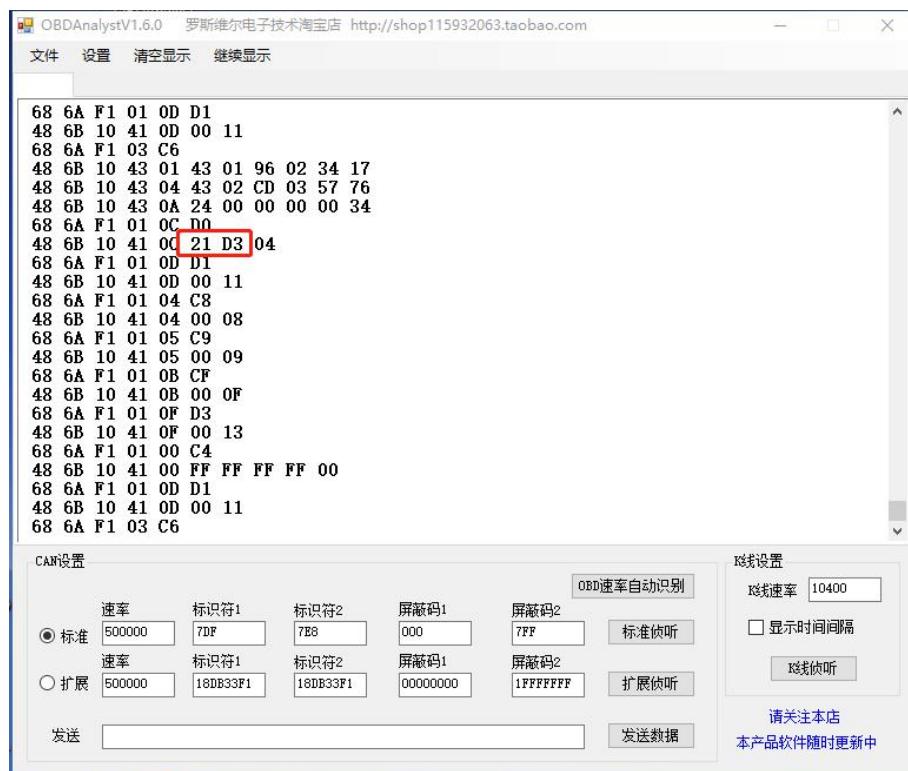
读数据流，此处，还是以发动机转速做一个举例，前面已经讲过很多例子了，发动机转速的分辨率就是 AB 字节除以 4 得到显示值。首先利用模拟器设置任意一个发动机转速，当前设置为 2164.75RPM。



C300 开发板读到的数据流发动机转速值如下图所示。



读取数据流发动机转速过程可以通过分析仪采集数据如下所示。



红色方框为发动机转速原始值 0x21d3, 可以转换为十进制值为 8659。发动机转速显示值 $8659/4=2164.75$ 。

7.4.2 ISO9141-2 协议在 Neulen TBOX C300 开发板中的程序实现

由于 ISO9141-2 协议程序源码基本与 ISO14230-4 协议的程序源码类似，这里将做一个简单介绍，详细细节可以参阅 7.3.2 小节内容。

初始化函数如下图所示。

```

24  /* ****
25  * @描述: ISO9141-2地址激活唤醒判断
26  * @参数: NONE
27  * @返回值: NL_OK:支持 NL_NOK:不支持
28  ****/
29  NLStatus ISO9141_2ADDR_WakeUp(void)
30  {
31      uint8_t i;
32      NLStatus err;
33      if(NL_OBD_Wakeup_Addr(0x33) == NL_OK)
34      {
35          err = NL_NOK;
36          for(i = 0; i < 3; i++)
37          {
38              NL_OBD_SendKLINEFrame(LinkCmd9141, Single, 800, &err);
39              _NL_Delay(1500);
40              if(err == NL_OK)
41              {
42                  break;
43              }
44          }
45          return err;
46      }
47      return NL_NOK;
48  }

```

33 行是 5 波特率初始化函数。相关初始化时序顺利完成后执行 36 行到 44 行代码，通过 for 循环执行 3 次数据请求，任意一次成功请求并获得汽车 ECU 响应，将退出循环，整个函数返回 NL_OK，否则返回 NL_NOK。

读车架号代码片段如下图所示。

```

257  if(OBDStruct.VINStruct.flag == RESET)
258  {
259      NL_ClearRAM((uint8_t*)OBDStruct.VINStruct.VIN, 18);
260      ram = ISO9141_2_GetVIN(&err);
261      if(err == NL_OK)
262      {
263          strncpy(OBDStruct.VINStruct.VIN, (const char*)ram, 17);
264          OBDStruct.VINStruct.flag = SET;
265      }
266  }

```

257 行是读车架号功能同步变量。260 行获取车架号内容，具体执行过程代码将在下图中展示。261 行到 265 行把读到的车架号内容拷贝到 OBDStruct.VINStruct.VIN 变量中，并设置读车架号功能同步变量 OBDStruct.VINStruct.flag 为 SET。

下面是函数 ISO9141_2_GetVIN 具体实现代码。

```

51 /* @描述: ISO9141-2获取车架号
52 * @参数: NONE
53 * @返回值: 车架号存储指针
54 */
55
56 char* ISO9141_2_GetVIN(NLStatus *err)
57 {
58     uint8_t i,j;
59     KLINERxRAMDef *ram;
60     ram = NL_OBD_SendKLINEFrame(VINCmd9141,Multi,800,err);
61     _NL_Delay(150);
62     if(*err == NL_OK)
63     {
64         for(j = 0; j < ram->Count; j++)
65         {
66             for(i = 0; i < 4; i++)
67             {
68                 VINRAM[i+j*4] = ram->RxRAM[j][6+i];
69             }
70         }
71     }
72     VINRAM[17] = 0;
73     return VINRAM;
74 }

```

函数中，60 行通过 `NL_OBD_SendKLINEFrame` 函数发送车架号请求。如果获得响应，62 行到 71 行把响应的数据提取其中的车架号内容存储在 `VINRAM` 数组中，73 行返回该数组地址。

读故障码程序片段如下图所示。

```

267 if(ISO9141_2_GetNotDrivingState() == NL_OK && OBDStruct.DTCStruct.flag == RESET)
268 {
269     ram = ISO9141_2_GetDTC(&err);
270     if(err == NL_OK)
271     {
272         NL_ClearRAM((uint8_t*)OBDStruct.DTCStruct.DTC,100);
273         strcpy(OBDStruct.DTCStruct.DTC,ram);
274         OBDStruct.DTCStruct.flag = SET;
275     }
276 }

```

267 行判断当前车速是否为 0，且读故障码同步变量是否为 `RESET`，条件满足执行 269 行读取当前车辆故障码内容，具体执行过程将在下图展示。读取故障码成功执行 270 到 276 行，把获得的故障码内容存储在 `OBDStruct.DTCStruct.DTC` 变量中，并设置读故障码同步变量为 `SET`。

下图展示 `ISO9141_2_GetDTC` 函数具体执行过程。

```

75  /*************************************************************************/
76  * @描述: ISO9141-2读取故障码
77  * @参数: *err:NL_OK:成功 NL_NOK:不成功
78  * @返回值: 故障码存储指针
79  *****/
80  char* ISO9141_2_GetDTC(NLStatus *err)
81  {
82      uint8_t i,j;
83      KLINERxRAMDef *ram;
84      uint16_t dtc;
85      NL_ClearRAM((uint8_t*)DTCRAM,200);
86      ram = NL_OBD_SendKLINEFrame(DTCCmd9141,Multi,800,err);
87      _NL_Delay(150);
88      if(*err == NL_OK)
89      {
90          for(j = 0; j < ram->Count; j++)
91          {
92              for(i = 0; i < 3; i++)
93              {
94                  if(ram->RxRAM[j][4+2*i] != 0 && ram->RxRAM[j][5+2*i] != 0)
95                  {
96                      dtc = ram->RxRAM[j][4+2*i]<<8|ram->RxRAM[j][5+2*i];
97                      strncpy(DTCRAM+strlen(DTCRAM),(const char*)NL_PCBU(dtc),5);
98                      strcpy(DTCRAM+strlen(DTCRAM),"");
99                  }
100             else
101             {
102                 DTCRAM[strlen(DTCRAM)-1] = 0;
103                 return DTCRAM;
104             }
105         }
106     }
107     DTCRAM[strlen(DTCRAM)-1] = 0;
108     return DTCRAM;
109 }
110 }
```

86 行通过 `NL_OBD_SendKLINEFrame` 函数发送故障码请求数据，如果获得汽车 ECU 正响应执行 88 到 107 行从响应数据中提取故障码存储于数组 `DTCRAM` 中，函数最后返回数组 `DTCRAM` 地址。

读数据流代码片段如下图所示。

```

277     if(OBDStruct.DSStruct.flag == RESET)
278     {
279         dsram = ISO9141_2_GetDS(ISODSItem,&err);
280         if(err == NL_OK)
281         {
282             OBDStruct.LINKSTATUS = SET;
283             OBDStruct.DSStruct.Total = dsram->Total;
284             for(i = 0; i < OBDStruct.DSStruct.Total; i++)
285             {
286                 strncpy(OBDStruct.DSStruct.DS[i],dsram->DS[i],30);
287             }
288             OBDStruct.DSStruct.flag = SET;
289         }
290         else
291         {
292             OBDStruct.LINKSTATUS = RESET;
293             NL_LED_ONOFF(LEDOBD,OFF,Fashing,LED1HZ);
294             break;
295         }
296     }
```

277 行是读数据流功能同步变量。279 行读数据流函数返回参数 `ISODSItem` 指示读取的数据流内容。如果读取成功执行 280 行到 289 行把读到的数据流存储在数组 `OBDStruct.DSStruct.DS` 中，读取失败则执行 291 行到 295 行，代码关闭 `OBD` 灯，并退出诊断功能。

下面是 `ISO9141_2_GetDS` 函数的具体代码截图。

```

138 //*****
139 * @描述: ISO9141-2读取数据流
140 * @参数: uint8_t *item: 数据流项目索引 ,*err :NL_OK:成功 NL_NOK:不成功
141 * @返回值: 数据流存储指针
142 *****/
143 DSStructDef* ISO9141_2_GetDS(DSItemStructDef item,NLStatus *err)
144 {
145     uint8_t i;
146     KLINERxRAMDef ram;
147     DSStruct .Total = item.Total;
148     for(i = 0; i < item.Total; i++)
149     {
150         DSCmd9141[5] = DSControl15301[item.Item[i]].PIDByte;
151         ram = NL_OBD_SendKLINEFrame(DSCmd9141,Single,800,err);
152         _NL_Delay(150);
153         if(*err == NL_OK)
154         {
155             ErrorCount = 0;
156             if (DSControl15301[item.Item[i]].Type == Numeric)
157             {
158                 sprintf(DSStruct.DS[i],DSControl15301[item.Item[i]].Format,DSControl15301[item.Item[i]].Equation0(ram->RxRAM[0]+DSControl15301[item.Item[i]].FineByte+2));
159             }
160             else
161             {
162                 strcpy(DSStruct.DS[i],"");
163                 strcpy(DSStruct.DS[i],DSControl15301[item.Item[i]].Equation1(ram->RxRAM[0]+DSControl15301[item.Item[i]].FineByte+2));
164                 strcpy(DSStruct.DS[i]+strlen(DSStruct.DS[i]),"");
165             }
166         }
167         else
168         {
169             if (++ErrorCount > 4)
170             {
171                 *err = NL_NOK;
172                 return 4DSStruct;
173             }
174         }
175     }
176 }
177
178 return &DSStruct;
179 }

```

148 行循环一次读取一个数据流，读取的数量和具体数据流由参数 item 决定。150 行根据 item 替换请求数据中 PID 的值。151 行发送请求数据给汽车，如果获得汽车 ECU 积极响应执行 153 到 167 行代码，其中 156 行到 160 行处理数值型数据流，162 行到 166 行处理字符型数据流，最后数据流结果存储在结构体变量 DSStruct 中。如果请求数据流数据没有获得响应，执行 169 行到 175 行代码，代码中进行未响应计数处理，计数超过 5，则认为与汽车 ECU 失去通信。函数最后返回结构体变量 DSStruct 地址。如上只是对 ISO9141-2 协议代码做了简要介绍，因为代码与 ISO14230-4 协议代码基本相同。在本节视频教程中，我会通过在线仿真形式对代码做具体介绍。

7.4.3 ISO9141-2 协议以及在开发板中的程序实现视频教程

7.4.1 ISO9141-2 协议解读视频教程

视频教程链接: <https://www.bilibili.com/video/BV1iE411H7dK?p=16>

7.4.2 ISO9141-2 协议在 Neulen TBOX C300 开发板中的程序实现

视频教程链接: <https://www.bilibili.com/video/BV1iE411H7dK?p=17>