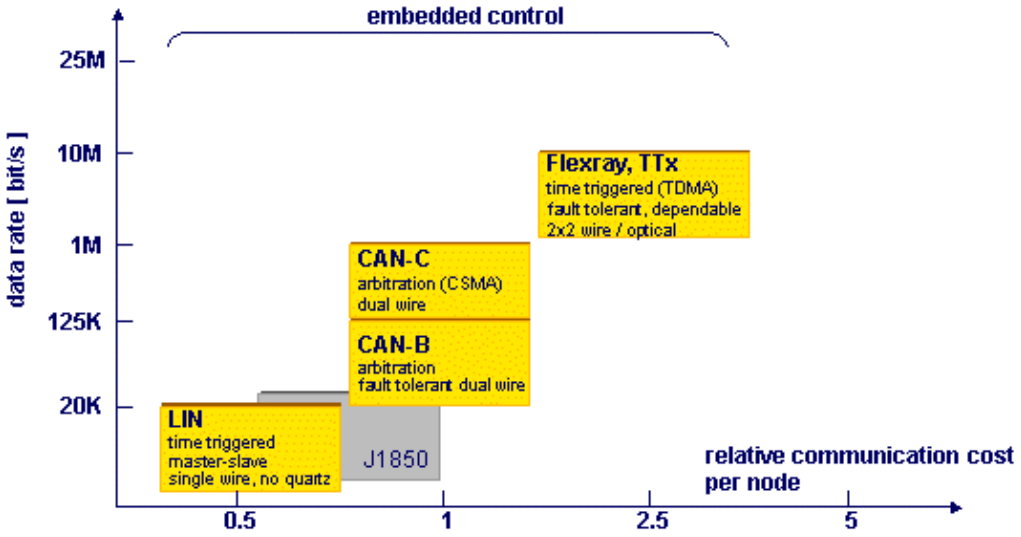
LIN 规范（V1.2）

LIN 标准

LIN 是低成本网络中的汽车通讯协议标准。



1 汽车中的主要网络协议

LIN 概念

LIN（Local Interconnect Network）是低成本的汽车网络，它是现有多种汽车网络在功能上的补充。由于能够提高质量、降低成本，LIN 将是在汽车中使用汽车分级网络的启动因素。LIN 的标准化将简化多种现存的多点解决方案，且将降低在汽车电子领域中的开发、生产、服务和后勤成本。

LIN 标准包括传输协议规范、传输媒体规范、开发工具接口规范和用于软件编程的接口。LIN 在硬件和软件上保证了网络节点的互操作性，并有可预测 EMC 的功能。

这个规范包包括了 3 个主要部分：

LIN 协议规范部分——介绍了 LIN 的物理层和数据链路层。

LIN 配置语言描述部分——介绍了 LIN 配置文件的格式。LIN 配置文件用于配置整个网络，并作为 OEM 和不同网络节点的供应商之间的通用接口，同时可作为开发和分析工具的一个输入。

LIN API 部分——介绍了网络和应用程序之间的接口。

这个概念可以实现开发和设计工具之间的无缝连接，并提高了开发的速度，增强了网络的可靠性。

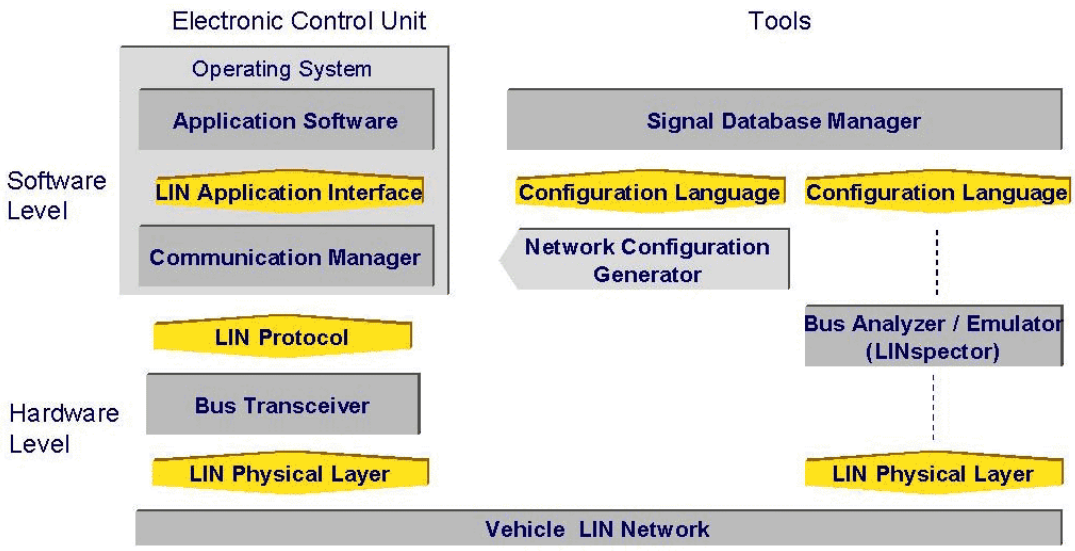


图 2 LIN 规范的范围

各部分链接

第一部分	LIN 协议规范
第二部分	LIN 配置语言规范
第三部分	LIN API 操作规程建议

LIN 协议规范

目录

1．介绍 2
1.1 修订历史 2
1.2 投稿人 2
2．基本概念 3
3．报文传输 7
3.1 报文帧 7
3.1.1 字节场（BYTE fields） 7
3.1.2 报头场（HEADER fields） 7
3.2 保留的标识符 10
3.3 报文帧的长度和总线睡眠检测 11
3.4 唤醒信号 11
4．报文滤波 12
5．报文确认 12
6．错误和异常处理 13
6.1 错误检测 13
6.2 错误标定 13
7．故障界定 13
8．振荡器容差 14
9．位定时要求和同步过程 14
9.1 位定时要求 14
9.2 同步过程 14
10．总线驱动器 / 接收器 15
10.1 总体配置 15
10.2 信号规范 15
10.3 线的特性 17
10.4 ESD/EMI 的符合条件 17
11．参考文献 18
A 附录 19
A.1 报文序列的举例 19
A.1.1 周期性的报文传输 19
A.1.2 总线唤醒过程 19
A.2 ID 场有效值表 19
A.3 校验和计算举例 21
A.4 报文错误的原因 21
A.5 故障界定的建议 22
A.5.1 主机控制单元 22
A.5.2 从机控制单元 22
A.6 物理接口的电源电压定义 23

1．介绍

LIN（Local Interconnect Network）是一个串行通讯协议，它有效地支持汽车应用中分布式机械电子节点的控制。它的使用范围是带单主机节点和一组从机节点[1]的 A 类多点总线。

LIN 总线的主要特性有：

- 单主机 / 多从机概念
- 基于普通 UART/SCI 接口的低成本硬件实现，低成本软件或作为纯状态机
- 从机节点不需要石英或陶瓷谐振器可以实现自同步
- 保证信号传输的延迟时间
- 低成本的单线设备
- 速度高达 20kbit/s

本规范的目：根据 ISO/OSI 参考模型的数据链路层和物理层，实现任何两个 LIN 设备的互相兼容（见图 2.1）。

LIN 是一个值得投资的总线通信，它不要求有 CAN 的带宽和多功能性。线驱动器 / 接收器的规范遵从 ISO 9141 标准[2]，而且 EMI 性能有所提高。

1.1 修订历史

1999 年 6 月 5 日： 修订版 1.0

2000 年 4 月 17 日： 修订版 1.1

2000 年 11 月 17 日： 修订版 1.2

- 协议
  - 表 2.1： 纠正发送速率单元
  - 第 2 章： 连接：把终端阻抗从范围值该成通常值
  - 表 3.1： 加入通常值列
  - 3.1.3 节： 确定响应场、校验和字节功能的使用
  - 3.2 节： 为总命令、服务报文及为以后扩展的 LIN 修订版（向上兼容）保留额外的标识符；命令报文代替前面的睡眠模式报文。
  - 3.3 节： 确定帧长度的计算
  - 表 3.4： 加入通常值列
  - 6.1 节： 删除标识符奇偶错误的错误处理
  - 6.2 节： 纠正校验和错误
  - 表 8.1： 新加规定带谐振器的主机和从机节点时钟容差
- 物理层
  - 表 10.3： 指定的最大翻转率
  - 表 10.4： 改变 C_{SLAVE} 和 C_{MASTER}，以取得更好的 ESD 和 EMI 性能
  - 10.4 节： 注意 ESD 电平的修改
- 整个文档： 用“报文帧”代替“数据帧”，或用更合适的名词。

1.2 投稿人

这个规范是由以下人员投稿：

J. Bauer, V. Riebeling, Audi AG, Ingolstadt.

J. Fröschl, M. Kaindl, Dr. J. Krammer, BMW AG, Munich.

C. Bracklo, W. Welja, DaimlerChrysler AG, Stuttgart.

R. Erckert, Dr. J. Krücken, Dr. A. Krüger, Dr. W. Specks, H.-C. Wense,

Motorola GmbH, Munich.

I. Horváth, A. Rajnák, Volcano Communications Technologies, Gothenburg.
J. Ende, T. Zawade, Volkswagen AG
L. Casparsson, Volvo Car Corporation, Gothenburg.

2．基本概念

LIN 协议有下面的特性：

- 单主机多从机组织（即没有总线仲裁）
- 保证信号传输的延迟时间
- 可选的报文帧长度：2，4 和 8 字节
- 配置的灵活性
- 带时间同步的多点广播接收，从机节点无需石英或陶瓷谐振器
- 数据校验和的安全性，和错误检测
- 检测网络中的故障节点
- 使用最小成本的半导体元件（小型贴片，单芯片系统）

根据 OSI 参考模型的 LIN 分层结构在图 2.1 中显示。

- 物理层定义了信号如何在总线媒体上传输。本规范中定义了物理层的驱动器 / 接收器特性。
- MAC（媒体访问控制）子层是 LIN 协议的核心。它管理从 LLC 子层接收到的报文，也管理发送到 LLC 子层的报文。MAC 子层由故障界定这个管理实体监控。
- LLC（逻辑链路控制）子层涉及报文滤波和恢复管理的功能。

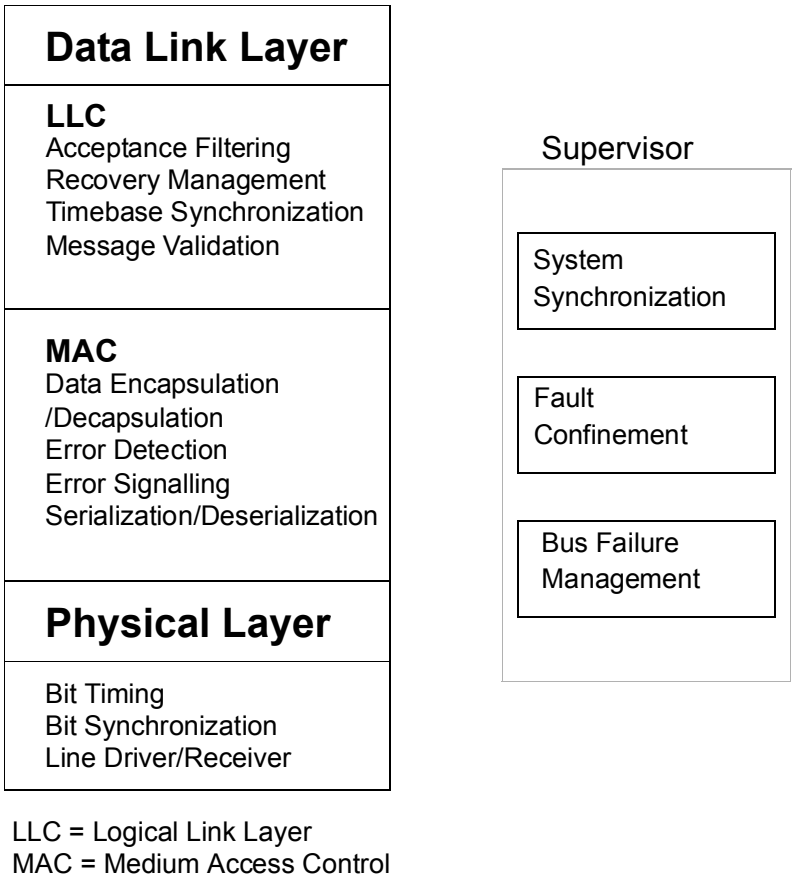


图 2.1 OSI 参考模型

这个规范的范围是定义数据链路层和物理层以及周围层上的 LIN 协议。

报文

在总线上发送的信息，有长度可选的固定格式（见第 3 章）。每个报文帧都包含 2、4 或 8 字节的数据以及 3 字节的控制、安全信息。总线的通讯由单个主机控制。每个报文帧都用一个分隔信号起始，接着是一个同步场和一个标识符场，这些都由主机任务发送。从机任务则是发回数据场和校验场（见图 2.2）。

通过主机控制单元中的从机任务，数据可以被主机控制单元发送到任何从机控制单元。相应的主机报文 ID 可以触发从机一从机的通信。

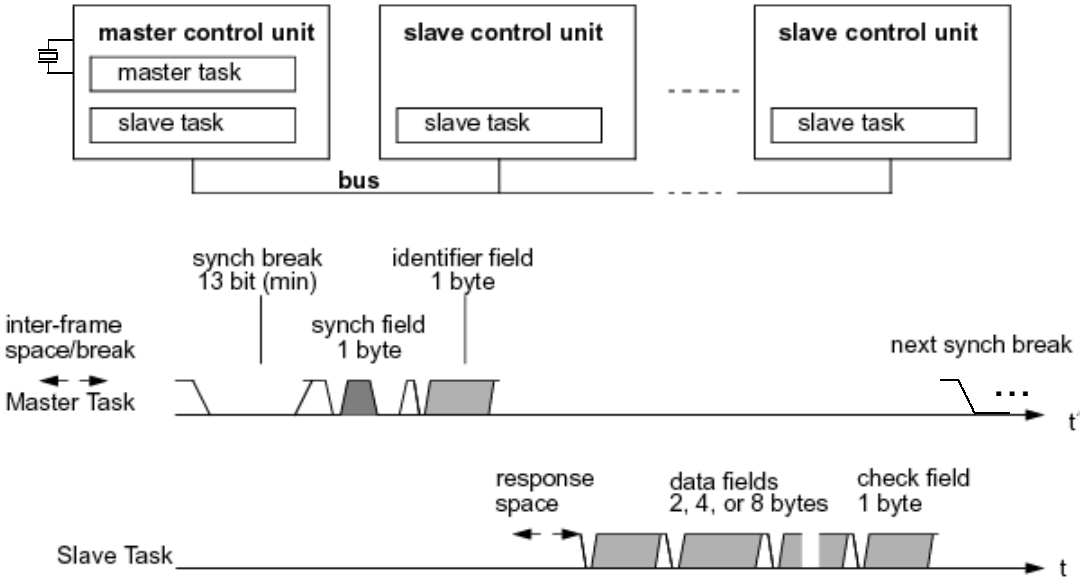


图 2.2 LIN 的通讯概念

信息路由

LIN 系统中，节点不使用有关系统配置的任何信息，除了单主机节点的命名。

系统灵活性：不需要改变任何其他从机节点的软件或硬件，就可以在 LIN 网络中直接添加节点。

报文路由：报文的内容由识别符命名。识别符不指出报文的目的地，但解释数据的含义。最大的标识符数量是 64，其中 4 个保留用于专用的通讯，譬如软件升级或诊断。

多播：由于引入了报文滤波的概念，任何数目的节点都可以同时接收报文，并同时对此报文做出反应。

位速率

最大的波特率是 20kbit/s，它是由单线传输媒体的 EMI 限制决定。最小的波特率是 1kbit/s，可以避免和实际设备的超时周期冲突。

为使用低成本的 LIN 器件，建议使用下面的位速率：

表 2.1 建议的位速率

低速	中速	高速
2400 bit/s	9600 bit/s	19200 bit/s

单主机－无仲裁

只有包含主机任务的控制器节点可以传输报文头，一个从机任务对这个报文头作出响应。由于没有仲裁过程，如果多于一个从机回应，则将产生错误。这种情况下的错误界定可由用户按照应用要求指定。

安全性

错误检测

- 监控，发送器比较总线“应当”的值和“现在”的值
- 数据场的校验和以 256 为模并取反，将 MSB 的进位加到 LSB 上
- 标识符场的双重奇偶校验保护

错误检测的性能

- 发送器可以检测到所有的本地错误
- 对整个协议的错误有很高的错误检出率

错误标定和恢复时间

单主机的概念中不允许进行直接的错误标定。错误在本地被检测到，并用诊断的形式请求（见第 6 章）。

故障界定

LIN 节点可以区分短时扰动和永久故障，它还能对故障作出合适的本地诊断和采取合适的行动（见第 7 章）。

连接

LIN 网络节点的最大数量不仅由标识符的数量限制（见上面的信息路由）也由总线的物理特性限制。

- 建议：LIN 网络的节点数量不应超过 16。否则，节点增加将减少网络阻抗，会导致环境条件变差，禁止无错误的通讯。每一个增加的节点都可以减少约 3%的网络阻抗（30kΩ||～1kΩ）。
- 网络中总的“电”线（通讯导线）长度应少于或等于 40m。
- 主机节点的总线端电阻典型值是 1kΩ，从机节点是 30kΩ。

单通道

总线有一个传送位的单通道。从这里数据可以获得数据的重新同步信息。

物理层

物理层是一条单线，每个节点通过上拉电阻线与总线，电源从汽车电源网络获得（V_{BAT}），见第 10 章。和上拉电阻串联的二极管可以防止电子控制单元（ECU）在本地电池掉电的情况下通过总线上电。

信号的波形由 EMI 和时钟同步的要求定义。

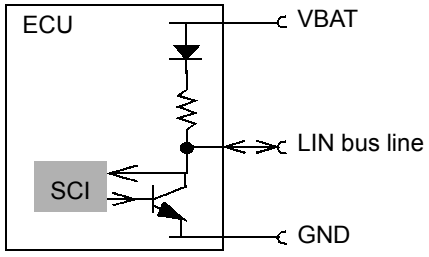


图 2.3 物理层的示意图

总线值

总线有两个互补的逻辑值：“显性”或“隐性”。相应的位值和电压值可参见表 2.2。

表 2.2 逻辑和物理总线值

逻辑值	位值	总线电压（见 10.2 章）
显性	0	地
隐性	1	电池

应答

正确接收报文后的应答过程在 LIN 协议中没有定义。主机控制单元检查由主机任务初始化的报文和由它自己的从机任务接收的报文的一致性。如果不一致（例如：丢失从机响应，校验和不正确等等），主机任务可以改变报文的进度表。

如果从机检测到不一致，从机控制器将保存这个信息并将它用诊断信息的形式向主机控制单元请求。诊断信息可按普通报文帧的形式进行发送。

命令帧和扩展帧

4 个 8 字节响应的标识符被保留用作特殊的报文帧：两个命令帧和两个扩展帧。

两个命令帧都包括 8 字节响应，可以用于从主机向从机节点（或相反）上载和下载数据。这个特征用于软件升级，网络配置和诊断。帧的结构和普通的报文相同。响应场包含用户定义的命令场而不是数据场，举个例子，命令场可以使从机节点进入服务模式或睡眠模式。

保留两个扩展帧标识符，用于将用户定义的报文格式和以后的 LIN 格式嵌入到现在的 LIN 协议中，而不需要改变当前的 LIN 规范。这就保证了 LIN 从机向上兼容以后的 LIN 协议修订版。扩展帧标识符向所有的总线成员声明了一个未定义的帧格式。标识符后面紧跟着的是 LIN 字节场的仲裁号码。接收到这个标识符的从机必须忽略后面的字节场，直到出现下一个同步间隔（synchronization break）。

睡眠模式 / 唤醒

为了减少系统的功耗，LIN 节点可以进入没有任何内部活动和被动总线驱动器的睡眠模式。用于广播睡眠模式的报文是一个专用的命令，在 3.2 节中定义。睡眠模式时，总线呈隐性。

任何总线活动或任何总线节点的内部条件都将结束（唤醒）睡眠模式。一旦节点被内部唤醒，基于唤醒信号的过程将给主机通报这一消息。唤醒帧是一个不变的显性位序列，参见 3.4 节。

唤醒后，内部的活动将重新启动，MAC 子层将等待系统振荡器稳定，从机节点则在重新参与总线通讯前等待，直到（自己）和总线活动同步（等待显性的同步间隔）。

时钟恢复和 SCI 同步

每个报文帧都由一个同步间隔（SYNCH BREAK）起始，接着是同步场（SYNCH FIRLD），这个同步场在几倍的位定时长度中包含了 5 个下降沿（即：“隐性”到“显性”的转换）。这个长度可以测量（即：通过定时器的捕获功能）而且可以用于计算从机节点内部定时（见 3.1 节和第 9 章）。

同步间隔帧将使能丢失了同步的从机节点识别同步场（见 3.1.2 节）。

振荡器容差

位定时的要求允许在有容差的从机节点上使用预设定的在片振荡器（参见表 8.1）。主机节点的时钟由石英或陶瓷谐振器发生，而且是“频率中心点”。

3. 报文传输

3.1 报文帧

报文传输是由报文帧的格式形成和控制。报文帧由主机任务向从机任务传送同步和标识符信息，并将一个从机任务的信息传送到所有其他从机任务。主机任务位于主机节点内部，它负责报文的进度表、发送报文头（HEADER）。从机任务位于所有的（即主机和从机）节点中，其中一个（主机节点或从机节点）发送报文的响应（RESPONSE）。

一个报文帧（见图 3.1）是由一个主机节点发送的报文头和一个主机或从机节点发送的响应组成。报文帧的报文头包括一个同步间隔场（SYNCH BREAK FIELD）、一个同步场（SYNCH FIELD）和一个标识符场。报文帧的响应（RESPONSE）则由 3 个到 9 个字节场组成：2、4 或 8 字节的数据场（DATA FIELD）和一个校验和场（CHECKSUM FIELD）。字节场由字节间空间分隔，报文帧的报文头和响应是由一个帧内响应空间分隔。最小的字节间空间和帧内响应空间是 0。这些空间的最大长度由报文帧的最大长度 T_{FRAME_MAX} 限制，这个长度在表 3.3 中指出。

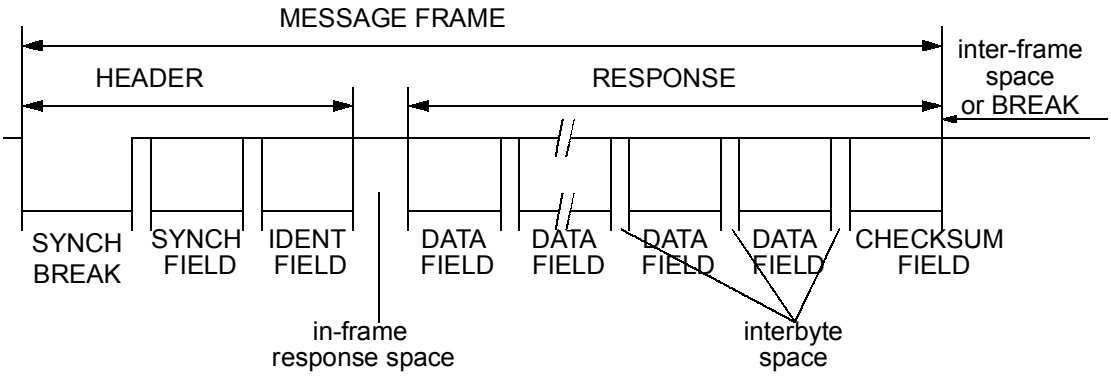


图 3.1 LIN 报文帧

3.1.1 字节场（BYTE fields）

字节场的格式（见图 3.2）就是通常的“SCI”或“UART”串行数据格式（8N1 编码）。每个字节场的长度是 10 个位定时（BIT TIME）。起始位（START BIT）是一个“显性”位，它标志着字节场的开始。接着是 8 个数据位，首先发送最低位。停止位（STOP BIT）是一个“隐性”位，它标志着字节场的结束。

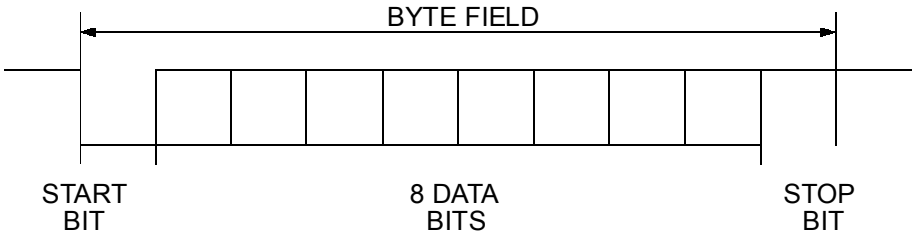


图 3.2 LIN 字节场

3.1.2 报文头场（HEADER fields）

同步间隔（SYNCHRONISATION BREAK）

为了能清楚识别报文帧的开始，报文帧的第一个场是一个同步间隔（SYNCH BREAK）。同步间隔场（SYNCH BREAK FIELD）是由主机任务发送。

它使所有的从机任务与总线时钟信号同步。

同步间隔场有两个不同的部分（见图 3.3）。第一个部分是由一个持续 T_{SYNBRK} 或更长时间（即最小是 T_{SYNBRK}，不需要很严格）的显性总线电平。接着的第二部分是最少持续 T_{SYNDEL} 时间的隐性电平，作为同步界定符。第二个场允许用来检测下一个同步场（SYNCH FIELD）的起始位。

最大的间隔和界定符时间没有精确的定义，但必须符合整个报文头 T_{HEADER_MAX} 的总体时间预算，T_{HEADER_MAX} 在表 3.3 中定义。

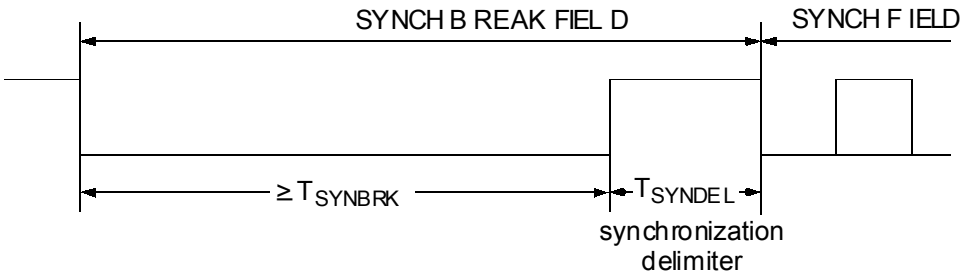


图 3.3 同步间隔场

同步间隔场（SYNCH BREAK FIELD）的位定时规范以及从机控制单元对此的估计值是考虑 LIN 网络中允许的时钟容差而得出的结果（见表 8.1）。如果显性电平持续的时间比在协议中定义的普通显性位序列（这里是：“0x00”场，有 9 个显性位）还要长，此时认为这是一个同步间隔场（SYNCH BREAK FIELD）。如果这个间隔超出了用从机位定时测量的间隔 T_{SBRKTS}，则从机节点将检测到一个间隔（见表 3.1）。这个“阀值”是由从机节点的最大本地时钟频率得出。基于精确的本地时基，阀值 T_{SBRKTS} 被指定了两个值。

同步间隔场（SYNCH BREAK FIELD）的显性电平长度至少为 T_{SYNBRK}（可以更长），这个时间是用主机位定时来测量。最小值应根据连接从机节点指定的最小本地时钟频率所要求的阀值而得出（见表 8.1）。

表 3.1 同步间隔场（SYNCH BREAK FIELD）的定时

同步间隔场	逻辑	名字	最小值[Tbit]	通常值[Tbit]	最大值[T_{bit}]
同步间隔低相位	显性	T_{SYNBRK}	13[a]		-
同步间隔界定符	隐性	T_{SYNDEL}	1[a]		-
同步间隔从机阀值	显性	T_{SBRKTS}		11[b]	
				9[c]	

a. 这个位定时基于主机的时基。

b. 这个位定时基于本地从机的位时基。它对时钟容差低于 $F_{TOL\_UNSYNCH}$ 的节点有效（见表 8.1），例如：有 RC 振荡器的从机节点。

c. 和 b 一样，但对对时钟容差低于 F_{TOL_SYNCH} 的节点有效，譬如带石英晶振或陶瓷谐振器的从机节点（见表 8.1）。

同步场（SYNCH FIELD）

同步场（SYNCH FIELD）包含了时钟的同步信息。同步场（SYNCH FIELD）的格式是“0x55”，表现在 8 个位定时中有 5 个下降沿（即：“隐性”跳变到“显性”的边沿）（见图 3.4）。同步的过程在第 9 章定义。

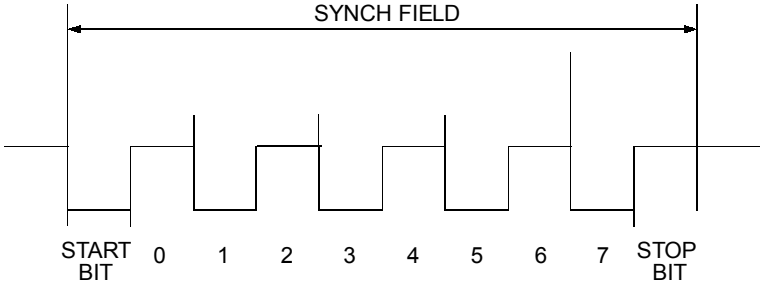


图 3.4 同步场

标识符场（IDENTIFIER FIELD）

标识符场（ID-FIELD）定义了报文的内容和长度。其中，内容是由 6 个标识符（IDENTIFIER）位和两个 ID 奇偶校验位（ID PARITY bit）表示（见图 3.5）。标识符位的第 4 和第 5 位（ID4 和 ID5）定义了报文的数据场数量 N_{DATA}（见表 3.2）。这将把 64 个标识符分成 4 个小组，每组 16 个标识符，这些标识符分别有 2，4 和 8 个数据场。

表 3.2 在报文帧中控制数据场数量

ID5	ID4	N_{DATA}（数据场的数量）[字节]
0	0	2
0	1	2
1	0	4
1	1	8

标识符有同样的 ID 位 ID0～ID3，但有不同的长度代码 ID4、ID5，可以表示不同的报文。

注意：如果在对此有严格的技术问题（譬如：在气象系统中）的系统中，报文的长度代码可以和表 3.2 中规定的不同。此时，数据字节的数量可以从 0～8 任意选择，而和标识符无关。

标识符的奇偶校验位通过下面的混合奇偶算法计算：

$P0 = ID0 \oplus ID1 \otimes ID2 \oplus ID4$ （奇校验）

$P1 = \overline{ID1 \oplus ID3 \otimes ID4 \oplus ID5}$ （偶校验）

这种情况下，不可能所有的位都是隐性或显性。

标识符 0x3C、0x3D、0x3E 和 0x3F 以及它们各自的标识符场 0x3C、0x7D、0xFE 和 0xBF（所有 8 字节报文）都保留用于命令帧（如：睡眠模式）和扩展帧（见 3.2 节）。

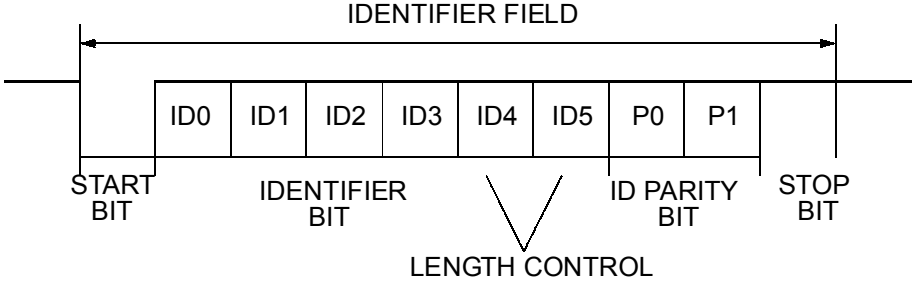


图 3.5 标识符场

3.1.3 响应场（RESPONSE field）

根据应用，如果信息和控制单元无关，则报文的响应场（数据、校验和）可以不需要处理，譬如不知道或错误的标识符。在这种情况下，校验和的计算可以忽略（参见附录 A.5）。

数据场（DATA FIELD）

数据场通过报文帧传输，由多个 8 位数据的字节场组成。传输由 LSB 开始（见图 3.6）。

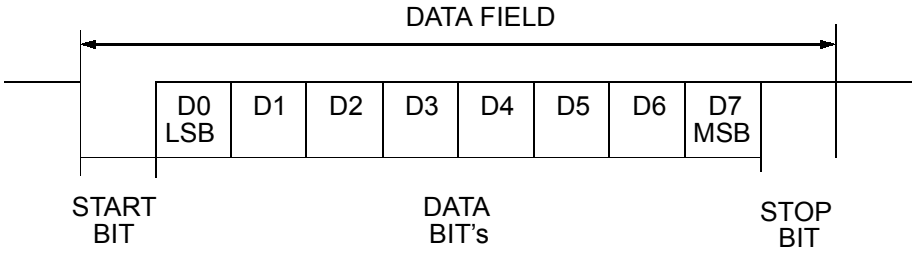


图 3.6 数据场

校验和场（CHECKSUM FIELD）

校验和场是数据场所有字节的和的反码（图 3.7）。和按“带进位加（ADDC）”方式计算，每个进位都被加到本次结果的最低位（LSB）。这就保证了数据字节的可靠性。

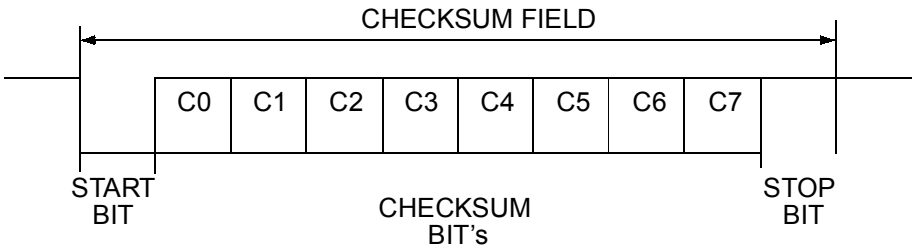


图 3.7 校验和场

所有数据字节的和的补码与校验和字节之加的和必须是“0xFF”。

3.2 保留的标识符

命令帧标识符（COMMAND FRAME IDENTIFIER）

保留的两个命令帧标识符用于主机向所有总线成员为服务广播普通命令请求。它的帧结构和普通的 8 位报文帧（见图 3.8）相同，只由保留的标识符来区别。

“0x3C” ID 场= 0x3C；ID0,1,6,7=0；ID 2,3,4,5=1 是一个主机请求帧，和

“0x3D” ID 场=0x7D；ID1,7=0；ID 0,2,3,4,5,6=1 是一个从机响应帧

（可参见附录 A2）。

标识符“0x3C”是一个“主机请求帧”（MasterReq），它可以从主机向从机节点发送命令和数据。标识符“0x3D”是一个“从机响应帧”（SlaveResp），它触发一个从机节点（由一个优先的下载帧编址）向主机节点发送数据。

保留第一个数据场为 0x00～0x7F 的命令帧，其用法由 LIN 协会定义。用户可以分配剩下的命令帧。

命令帧的第一个数据字节：D7 位＝0： 保留使用

D7 位＝1： 自由使用

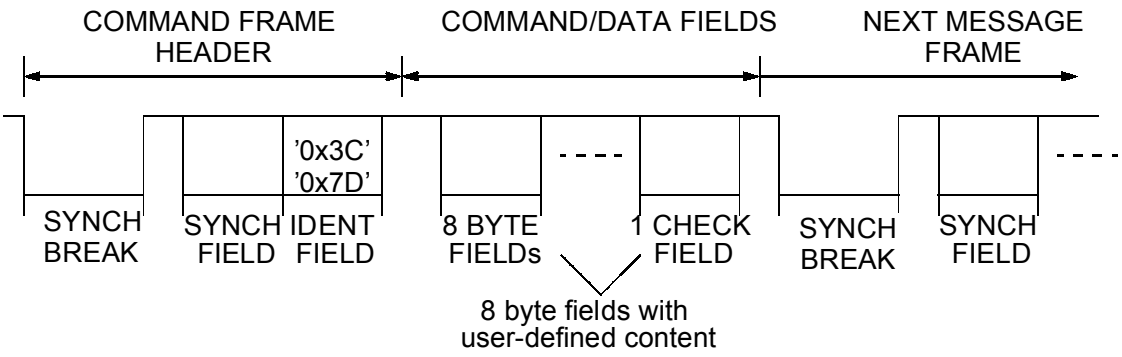


图 3.8 LIN 命令帧

睡眠模式命令

睡眠模式命令用于将睡眠模式广播到所有的总线节点。在完成这个报文后，一直到总线上出现唤醒信号结束睡眠模式前，将没有总线活动（见 3.4 节）。睡眠模式命令是第一个数据字节是 0x00 的下载命令帧。

扩展帧标识符

保留的两个扩展帧标识符允许在不改变现有 LIN 规范的情况下，在 LIN 协议中嵌入用户定义的报文格式或以后的 LIN 格式。这就保证了 LIN 从机可以向上兼容以后的 LIN 协议修订版。

扩展帧用保留的标识符场区别：

“0x3E” ID 场=0xFE；ID0=0；ID 1,2,3,4,5,6,7=1 是用户定义的扩展帧，和

“0x3F” ID 场=0xBF；ID6=0；ID 0,1,2,3,4,5,7=1 是以后的 LIN 扩展帧（参见附录 A2）

标识符“0x3E”（标识符场=“0xFE”）表示一个用户定义的扩展帧，它可被自由使用。标识符“0x3F”（标识符场=“0xBE”）直接保留给以后的 LIN[1]扩展版本，现在还不能使用。

标识符后面可以跟随任意数量的 LIN 字节场（见图 3.9）。这里没有定义帧的长度、通讯概念[2]和数据内容。ID 场的长度编码对这两个帧不起作用。

从机接收扩展帧标识符，但如果不使用它的内容，则必须忽略所有的后续 LIN 字节区直到接收到下一个同步间隔（SYNCH BREAK）。

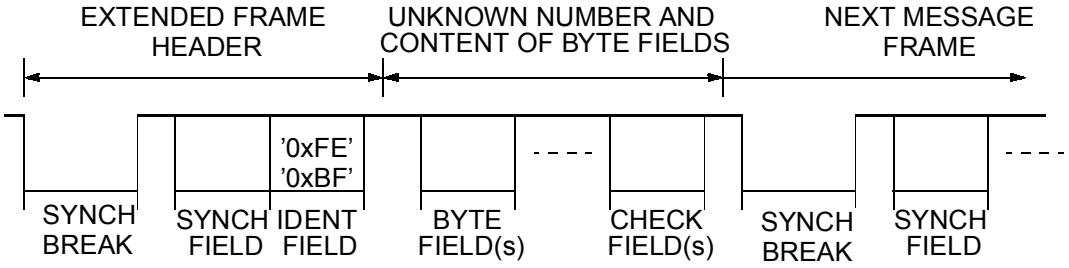


图 3.9 LIN 扩展帧

3.3 报文帧的长度和总线睡眠检测

报文帧用一个同步间隔场作为起始，用校验和场作为结束。报文帧中的字节场用字节间空间和帧内响应空间分隔。字节间空间和帧内响应空间的长度没有定义，只限制了整个报文帧的长度。最小的帧长度 T_{FRAME_MIN} 是传输一个帧所需要的最小时间（字节间空间和帧间响应空间是 0）。最大的帧长度 T_{FRAME_MAX} 是允许传输一个帧的最大时间。时间值请看表 3.3。它们由数据场字节 N_{DATA} 的数量决定，并不包括系统固有的（譬如：物理上）信号延时。

表 3.3 报文帧的定时

时间	名字	时间[T_{bit}]
最小报文帧长度	T_{FRAME_MIN}	$10*N_{DATA}+44$
最小报头长度	T_{HEADER_MIN}	34
最大报头长度	T_{HEADER_MAX}	（$T_{HEADER\_MIN}+1^a$）*1.4
最大报文帧长度	T_{FRAME_MAX}	（$T_{FRAME\_MIN}+1^a$）*1.4
总线空闲超时	T_{TIME_OUT}	25,000

a. “＋1”的条件使 T_{HEADER_MIN} 和 T_{FRAME_MAX} 是一个整数值

如果从机检测到总线在 T_{TIME_OUT} 中没有活动，它会假设总线处于睡眠模式。这也可能是由于睡眠报文被破坏。

3.4 唤醒信号

总线的睡眠模式可以通过任何节点发生一个唤醒信号来中止。唤醒信号可以通过任何从机任务发送，但只有总线以前处于睡眠模式且节点内部请求被挂起时才有效。

---

[1] 和 CAN 协议中的“标准帧”切换到“扩展帧”相比较[3]

[2] 甚至可以是多主机

唤醒信号是字符“0x80”。当从机不和主机节点同步时，信号可以比精确的时钟源信号拉长 15%或缩短 15%。主机可以检测到字符“0x80”，并作为一个有效的数据字节，“0xC0”、“0x80”或“0x00”都可以。第一个场由 T_{WUSIG} 的显性位序列给出，即 8 个显性位（包括起始位）。接着的第二个场是持续了至少 T_{WUDEL} 的隐性唤醒界定符，即至少 4 个位定时（包括停止位和一个隐性暂停位）。

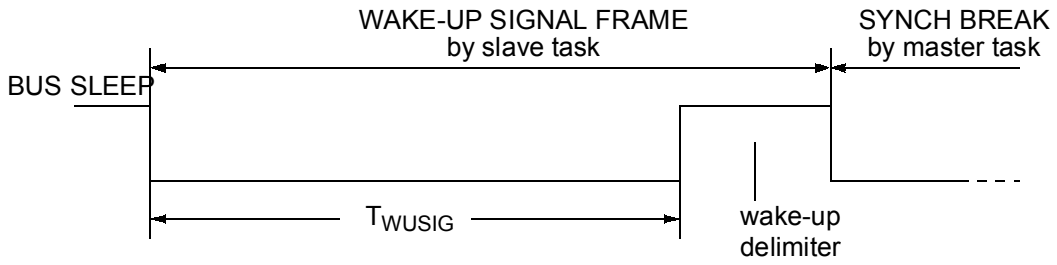


图 3.10 唤醒信号帧

在唤醒信号发送到总线上后，所有的节点都运行启动过程，并等待主机任务发送一个同步间隔场和同步场。如果在唤醒信号超时（TIME-OUT AFTER WAKEUP SIGNAL）时间内没有检测到同步场，请求第一个唤醒信号的节点将再一次发送一个新的唤醒信号。但这种情况将不超过 3 次。然后唤醒信号的传输将被 3 个间隔超时（TIME-OUT AFTER THREE BREAKS）挂起，详细情况请看表 3.4 和附录 A.1。只有有内部唤醒请求挂起的节点才允许重新发送唤醒信号。在 3 个间隔超时后再重新发送 3 个唤醒信号，此后就可以决定是否要停止重新发送。

表 3.4 唤醒信号定时

唤醒	逻辑	名字	最小值[T_{bit}]	通常值[T_{bit}]	最大值[T_{bit}]
唤醒信号	显性	T_{WUSIG}		8a	
唤醒信号界定符	隐性	T_{WUDEL}	4b		64
唤醒信号超时	隐性	T_{TOBRK}			128
3 个间隔超时	隐性	T_{T3BRK}	15,000		

a. 这个位定时是基于各自的从机时钟。

b. 要检查这个唤醒时间对所有网络节点是否足够。

如果没有其他的节点，位定时 Tbit 参照主机节点的 SCI 波特率（见第 9 章）。

4．报文滤波

报文滤波是基于整个标识符。必须通过网络配置来确认：每一个从机任务对应一个传送标识符。

5．报文确认

如果直到帧的结尾都没有检测到错误，这个报文对发送器和接收器都有效。

如果报文发生错误，则主机和从机任务都认为报文没有发送。

注意：

主机和从机任务在发送和接收到一个错误报文时所采取的行动并没有在协议规范中定义。像主机重新发送或从机的后退操作都由应用的要求来决定，而且要在应用层中说明。

在总线上传送的事件信息也可能丢失，而且这个丢失不能被检测到。

6. 错误和异常处理

6.1 错误检测

这里共定义了 5 个不同的报文错误类型。产生错误的原因列在附录 A.4：

位错误

向总线发送一个位的单元同时也在监控总线。当监控到的位的值和发送的位的值不同时，则在这个位定时检测到一个位错误。

校验和错误

所有数据字节的和的补码与校验和字节之加的和不是“0xFF”时，则检测到一个校验和错误（见 3.1 节，校验和场）。

标识符奇偶错误

标识符的奇偶错误（即：错误的标识符）不会被标出。通常，LIN 从机应用不能区分一个未知但有效的标识符和一个错误的标识符。然而，所有的从机节点都能区分 ID 场中 8 位都已知的标识符和一个已知但错误的标识符。

从机不响应错误

如果任何从机任务在发送 SYNCH 和标识符场时，在最大长度时间 T_{FRAME_MAX}（见 3.3 节）中没有完成报文帧的发送，则产生一个不响应错误。

同步场不一致错误

当从机检测到同步场的边沿在给出的容差外，则检测到一个同步场不一致错误（见第 8 章）。

没有总线活动

如果在接收到最后一个有效信息后，在 $T_{TIMEOUT}$（见 3.3 节）的时间内没有检测到有效的同步间隔场或字节场，则检测到一个没有总线活动条件。

6.2 错误标定

LIN 协议不标定检测到的错误。错误由每个总线节点标记而且可以被第 7 章描述的故障界定过程访问。

7. 故障界定

故障界定的概念主要定位于使主机节点可以处理尽量多的错误检测、错误恢复和诊断。故障界定主要基于系统的要求，它除了一些很小的特征外都不是 LIN 协议的一部分。可能的错误原因请参看附录 A.4，附录 A.5 是建议的故障界定过程。

主机控制单元

主机控制单元要检测下面的错误状况：

- 主机任务发送：当回读自己的发送时，在同步或标识符字节检测到一个位错误或标识符奇偶错误。
- 主机控制单元中的从机任务接收：当从总线期望或读一个数据时，检测到一个从机不响应错误或校验和错误。

从机控制单元

任何从机控制单元要检测以下的错误情况：

- 从机任务发送：当回读自己的发送时，在数据或校验和场有位错误。

- 从机任务接收：当从总线读值时，检测到一个标识符奇偶错误和一个校验和错误。

当从总线上读值时，会检测到一个从机不响应错误。

当一个从机期望从另外一个从机（由标识符决定）接收报文但在报文帧的最大长度 T_{FRAME_MAX}（见表 3.3）的时间内总线上没有有效的报文，则产生错误，而且这个错误类型会被检测到。但当从机不准备接收报文（由标识符决定），它就不需要检测到这个错误。

当在给出的容差（见第 8 章）中没有检测到同步场的边沿，则检测到一个同步字节不一致错误。

8. 振荡器容差

在片时钟发生器使用内部校准时可以使频率容差比±15%更好。这个精度足以在报文流中检测到同步间隔。接着，使用同步场的精细校准可以确保适当地接收和发送报文。在考虑操作中的温度影响以及电压漂移的情况下，在片振荡器要在其余报文中保持稳定。

表 8.1 振荡器容差

时钟容差	名字	△F/FMaster
主机节点	$F_{TOL\_RES\_MASTER}$	<±0.5%
带石英晶振或陶瓷谐振器的从机节点（不需要同步）	$F_{TOL\_RES\_SLAVE}$	<±1.5%
没有谐振器的从机，丢失同步	$F_{TOL\_UNSYNCH}$	<±15%
没有谐振器的从机，同步并有完整的信息	F_{TOL_SYNCH}	<±2%

9. 位定时要求和同步过程

9.1 位定时要求

如果没有其他情况，本文档中的所有位时间都参考主机节点的位定时。

9.2 同步过程

同步场的模式是“0x55”。同步过程是基于模式下降沿之间的时间量度。下降沿在 2、4、6 和 8 位时间有效，可以简单地计算基本位时间 T_{bit}。

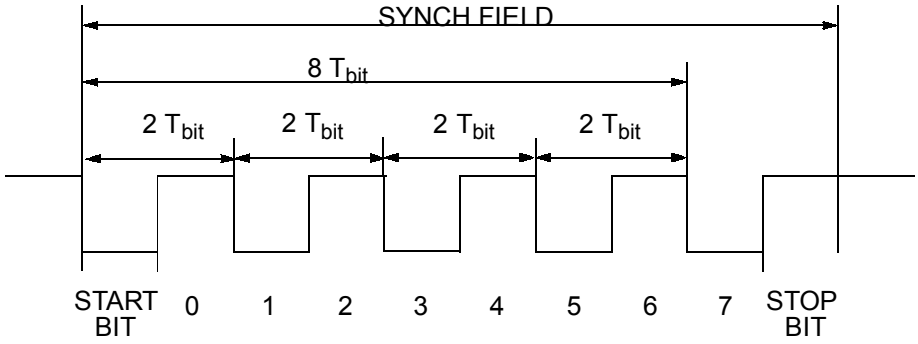


图 9.1 同步场

我们建议测量起始位和第 7 位下降沿之间的时间，并将得到的值除 8。将结果除 8 是将二进制的定时器值向 LSB 右移 3 位，将最低位四舍五入，校正即得到结果。

10．总线驱动器／接收器

10.1 总体配置

总线驱动器／接收器是一个 ISO 9141 标准的增强设备。它包括双向 LIN 总线，这个双向总线连接每个节点的驱动器／接收器，并通过一个终端电阻和一个二极管连接到电池节点的正极 V_{BAT}（见图 10.1）。二极管可以在“丢失电池”（掉电）的情况下，阻止 ECU 从总线不受控制地上电。

要注意：LIN 规范将电子控制单元（ECU）的外部电气连接电压作为参考电压，而不是将 ECU 内部电压作为参考电压。当设计 LIN 的收发器电路时，特别要考虑二极管的反向极性寄生电压降。

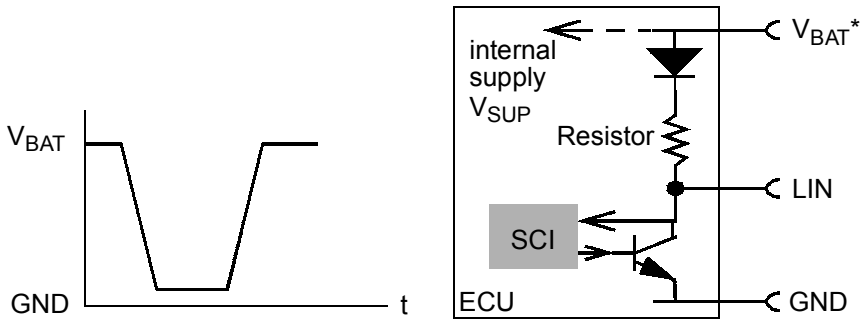


图 10.1 单线的汽车总线接口概念（*见附录 A.6）

10.2 信号规范

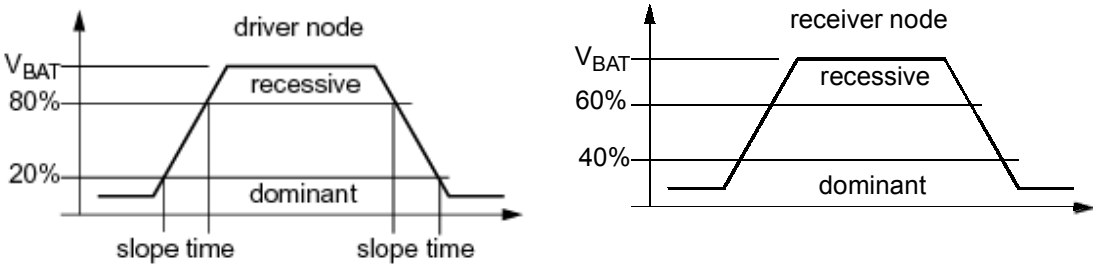


图 10.2 总线上的电压电平

LIN 物理层的电气直流参数和端电阻的值分别列在表 10.1 和表 10.2。注意，由于在一个集成的电阻／二极管网络中没有寄生的电流通路，所以要在总线和 ECU 内部电压（V_{SUP}）之间形成一条寄生电流通道，譬如通过 ESD 元件。

表 10.1 LIN 收发器的电子直流参数

参数	最小值	典型值	最大值	单位	备注
V_{BAT}[a]	8		18	V	工作电压范围
$V_{BAT\_NON\_OP}$	-0.3		40	V	器件不被破坏的电压范围
I_{BUS}[b]@V_{BUS}=1.2V	40		200	mA	显性状态（驱动器启动）c
I_{BUS}	-1.1*V_{BAT}/R				显性状态（驱动器关闭） R：=在表 10.2 中定义的上拉阻抗
I_{BUS}@V_{BUS}=V_{BAT} 8V<V_{BAT}<18V			20	μA	隐性状态。当 V_{BUS}>V_{BAT} 也可以应用

I_{BUS}@-12V<V_{BUS}<0V 控制单元没有对地连接	-1		1	mA	丢失本地接地必须不影响剩下的网络通讯
I_{BUS}@-18V<V_{BUS}<-12V 控制单元没有对地连接					节点要维持这种情况下的电流。总线必须在这种情况下可工作。
V_{BUSdom}	-8		0.4*V_{BAT}	V	接收器显性状态
V_{BUSrec}	0.6*V_{BAT}		18	V	接收器隐性状态

a. V_{BAT} 表示控制单元连接器的电源电压，它可能和供给电子器件的内部电源 V_{SUP} 不一样（见附录 A.6）

b. I_{BUS}：流进节点的电流

c. 收发器必须可以下拉电流至少 40mA。流入节点的最大电流不能超过 200mA，以避免可能的损坏。

表 10.2 上拉电阻的参数

参数	最小值	典型值	最大值	单位	备注
R_{master}	900	1000	1100	Ω	必须要有串联二极管（图 10.1）
R_{slave}	20	30	47	KΩ	必须要有串联二极管

LIN 物理层的电气 AC 交流参数在表 10.3 列出，定时参数在图 10.3 定义。

表 10.3 LIN 物理层的电气 AC 交流参数

参数	最小值	典型值	最大值	单位	备注
|dV/dt| 上升和下降沿（旋转率）	1	2	3	V/μs	LIN 总线的 EMI 特性由信号的旋转率决定，譬如 di/dt 和 d2V/dt2 是因素之一。旋转率的值要接近 2V/μs，一方面可以减少辐射，另一方面允许传输速率高达 20kBit/sec。
t_{trans_pd} 发送器的传输延时			4	μs	见图 10.3 $t_{trans\_pd}=max(t_{trans\_pdr},t_{trans\_pdf})$
t_{rec_pd} 接收器的传输延时			6	μs	见图 10.3 $t_{rec\_pd}=max(t_{rec\_pdr},t_{rec\_pdf})$
t_{rec_sym} 接收器传输延时的上升沿和下降沿的对称度	-2		2	μs	见图 10.3 $t_{rec\_sym}=t_{rec\_pdf}-t_{rec\_pdr}$
t_{trans_sym} 发送器传输延时的上升沿和下降沿的对称度	-2		2	μs	见图 10.3 $t_{trans\_sym}=t_{trans\_pdf}-t_{trans\_pdr}$
t_{therm} 短路恢复时间	1.5			ms	在检测到短路后，发送器必须可以再次冷却。因此，发送器电路此时不能启动。

时序图：

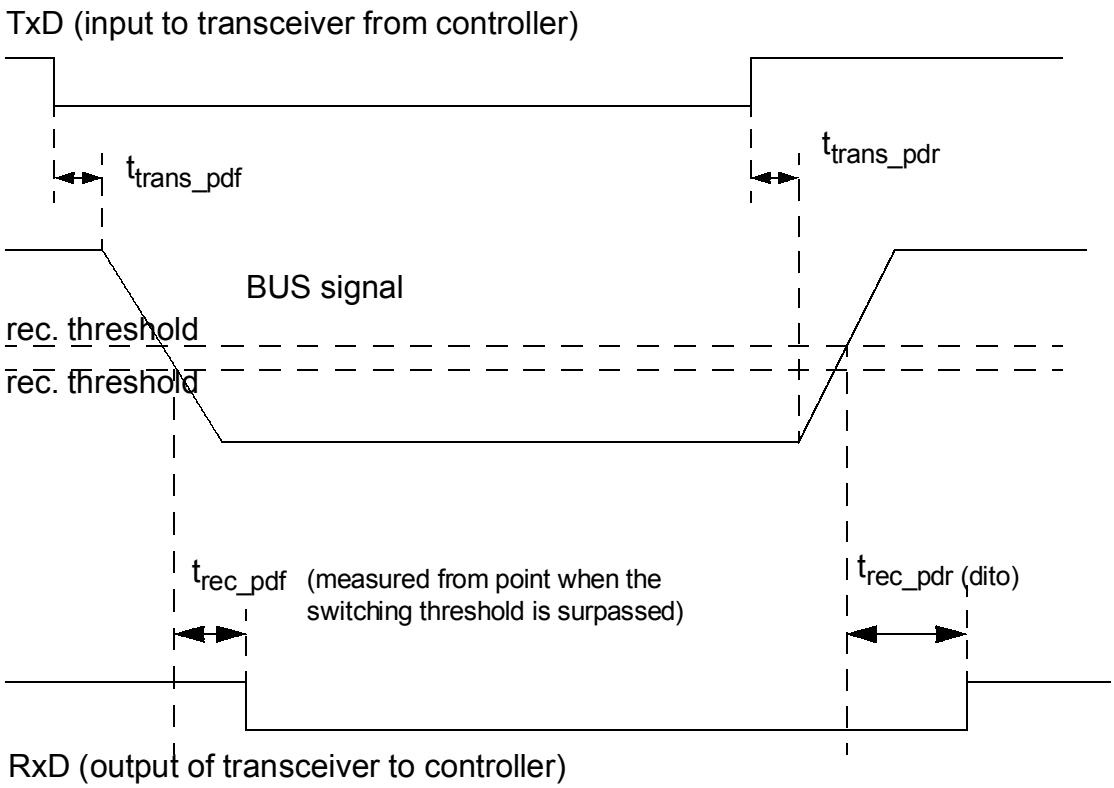


图 10.3 总线时序的定义

10.3 线的特性

总线信号上升和下降的最大旋转率实际上由典型总线收发器控制的旋转率限制。上升信号的最小旋转率由 RC 时间常数给定。因此，总线的电容应保持非常低，使波形的有大的非对称性。主机模块选择的电容要比从机模块大，这样可以作为不同数量的节点网络变量的“缓冲器”。整个总线的电容 C_{BUS} 可以用下面的方程 1 算出：

$$C_{BUS} = C_{MASTER} + n \cdot C_{SLAVE} + \overline{C}_{LINE} \cdot LEN_{BUS} \qquad \text{方程 1}$$

考虑表 10.4 给出的参数。

表 10.4 线的特性和参数

	名字	典型值	最大值	单位
总线的整个长度	LEN_{BUS}		40	m
包括从机和主机电容的整个总线电容量	C_{BUS}	4	10	nF
主机节点的电容量	C_{MASTER}	220	2500	pF
从机节点的电容量	C_{SLAVE}	220	250	pF
线电容	$\overline{C}_{LINE}$	100	150	pF/m

10.4 ESD/EMI 的符合条件

半导体物理层设备必须遵守根据 IEC 1000-4-2:1995 的要求，保护不受人体放电损坏。最小的放电电压级是±2000V。

注意：在 ECU 连接器的汽车应用中，要求的 ESD 电压级可达±8000V。

11．参考文献

[1] J.W. Specks, A, Rajnák, .LIN - Protocol, Development Tools, and Software Interfaces for Local Interconnect Networks in Vehicles., *9th Congress on Electronic Systems for Vehicles*, Baden-Baden, Germany, Oct. 5/6, 2000

[2] .Road vehicles - Diagnostic systems - Requirement for interchange of digita information., *International Standard ISO9141*, 1st Edition, 1989

[3] Robert Bosch GmbH, "CAN Specification", *Version 2.0, Part B*, Stuttgart, 1991

A 附录

A.1 报文序列的举例

A.1.1 周期性的报文传输

总线上通常的报文传输如下所示：

<MF 1> <IF-Space> <MF 2> <IF-Space> ... <IF-Space> <MF n> <IF-Space>

<MF 1> <IF-Space> <MF 2> <IF-Space> ... <IF-Space> <MF n> <IF-Space>

<MF 1> <IF-Space> <MF 2> <IF-Space> ... <IF-Space> <MF n> <IF-Space>

....

[MF = 报文帧（Message Frame）；IF-Space = 帧间空间（InterFrame Space）]

它可以预知最差情况的定时。

A.1.2 总线唤醒过程

在睡眠模式中，没有总线活动。任何从机节点可以发送一个唤醒信号中止睡眠模式。在普通的情况下，主机节点会用一个同步间隔启动报文的发送：

[SLEEP MODE] [NODE-INTERNAL WAKE-UP] <WAKE-UP SIGNAL>

<MF 1> <IF-Space> <MF 2> <IF-Space> ... <IF-Space> <MF n> <IF-Space>

<MF 1> <IF-Space> <MF 2> <IF-Space> ... <IF-Space> <MF n> <IF-Space>

....

如果主机节点没有响应，从机将最多再发送 2 次唤醒信号。然后，唤醒尝试将在某段时间内挂起，直到它恢复：

[SLEEP MODE] [NODE-INTERNAL WAKE-UP]

<WAKE-UP SIGNAL> <TIME-OUT AFTER BREAK>

<WAKE-UP SIGNAL> <TIME-OUT AFTER BREAK>

<WAKE-UP SIGNAL> <TIME-OUT AFTER THREE BREAKS>

[REPEAT BUS WAKE-UP PROCEDURE IF STILL PENDING]

A.2 ID 场有效值表

表 A.2.1 ID 场有效值

ID[0..5]		$P0 = ID0 \oplus ID1 \oplus ID2 \oplus ID4$	$P1 = \overline{ID1 \oplus ID3 \oplus ID4 \oplus ID5}$	ID 场	ID 场		数据字节数量
Dec	Hex			7 6 5 4 3 2 1 0	DEC	Hex	
0	0x00	0	1	1 0 0 0 0 0 0 0	128	0x80	2
1	0x01	1	1	1 1 0 0 0 0 0 1	193	0xC1	2
2	0x02	1	0	0 1 0 0 0 0 1 0	66	0x42	2
3	0x03	0	0	0 0 0 0 0 0 1 1	3	0x03	2
4	0x04	1	1	1 1 0 0 0 1 0 0	196	0xC4	2
5	0x05	0	1	1 0 0 0 0 1 0 1	133	0x85	2
6	0x06	0	0	0 0 0 0 0 1 1 0	6	0x06	2
7	0x07	1	0	0 1 0 0 0 1 1 1	71	0x47	2
8	0x08	0	0	0 0 0 0 1 0 0 0	8	0x08	2
9	0x09	1	0	0 1 0 0 1 0 0 1	73	0x49	2
10	0x0A	1	1	1 1 0 0 1 0 1 0	202	0xCA	2
11	0x0B	0	1	1 0 0 0 1 0 1 1	139	0x8B	2
12	0x0C	1	0	0 1 0 0 1 1 0 0	76	0x4C	2
13	0x0D	0	0	0 0 0 0 1 1 0 1	13	0x0D	2

14	0x0E	0	1	1 0 0 0 1 1 1 0	142	0x8E	2
15	0x0F	1	1	1 1 0 0 1 1 1 1	207	0xCF	2
16	0x10	1	0	0 1 0 1 0 0 0 0	80	0x50	2
17	0x11	0	0	0 0 0 1 0 0 0 1	17	0x11	2
18	0x12	0	1	1 0 0 1 0 0 1 0	146	0x92	2
19	0x13	1	1	1 1 0 1 0 0 1 1	211	0xD3	2
20	0x14	0	0	0 0 0 1 0 1 0 0	20	0x14	2
21	0x15	1	0	0 1 0 1 0 1 0 1	85	0x55	2
22	0x16	1	1	1 1 0 1 0 1 1 0	214	0xD6	2
23	0x17	0	1	1 0 0 1 0 1 1 1	151	0x97	2
24	0x18	1	1	1 1 0 1 1 0 0 0	261	0xD8	2
25	0x19	0	1	1 0 0 1 1 0 0 1	153	0x99	2
26	0x1A	0	0	0 0 0 1 1 0 1 0	26	0x1A	2
27	0x1B	1	0	0 1 0 1 1 0 1 1	91	0x5B	2
28	0x1C	0	1	1 0 0 1 1 1 0 0	156	0x9C	2
29	0x1D	1	1	1 1 0 1 1 1 0 1	221	0xDD	2
30	0x1E	1	0	0 1 0 1 1 1 1 0	94	0x5E	2
31	0x1F	0	0	0 0 0 1 1 1 1 1	31	0x1F	2
32	0x20	0	0	0 0 1 0 0 0 0 0	32	0x20	4
33	0x21	1	0	0 1 1 0 0 0 0 1	97	0x61	4
34	0x22	1	1	1 1 1 0 0 0 1 0	226	0xE2	4
35	0x23	0	1	1 0 1 0 0 0 1 1	163	0xA3	4
36	0x24	1	0	0 1 1 0 0 1 0 0	100	0x64	4
37	0x25	0	0	0 0 1 0 0 1 0 1	37	0x25	4
38	0x26	0	1	1 0 1 0 0 1 1 0	166	0xA6	4
39	0x27	1	1	1 1 1 0 0 1 1 1	231	0xE7	4
40	0x28	0	1	1 0 1 0 1 0 0 0	168	0xA8	4
41	0x29	1	1	1 1 1 0 1 0 0 1	233	0xE9	4
42	0x2A	1	0	0 1 1 0 1 0 1 0	106	0x6A	4
43	0x2B	0	0	0 0 1 0 1 0 1 1	43	0x2B	4
44	0x2C	1	1	1 1 1 0 1 1 0 0	236	0xEC	4
45	0x2D	0	1	1 0 1 0 1 1 0 1	173	0xAD	4
46	0x2E	0	0	0 0 1 0 1 1 1 0	46	0x2E	4
47	0x2F	1	0	0 1 1 0 1 1 1 1	111	0x6F	4
48	0x30	1	1	1 1 1 1 0 0 0 0	240	0xF0	8
49	0x31	0	1	1 0 1 1 0 0 0 1	177	0xB1	8
50	0x32	0	0	0 0 1 1 0 0 1 0	50	0x32	8
51	0x33	1	0	0 1 1 1 0 0 1 1	115	0x73	8
52	0x34	0	1	1 0 1 1 0 1 0 0	180	0xB4	8
53	0x35	1	1	1 1 1 1 0 1 0 1	245	0xF5	8
54	0x36	1	0	0 1 1 1 0 1 1 0	118	0x76	8
55	0x37	0	0	0 0 1 1 0 1 1 1	55	0x37	8
56	0x38	1	0	0 1 1 1 1 0 0 0	120	0x78	8

57	0x39	0	0	0 0 1 1 1 0 0 1	57	0x39	8
58	0x3A	0	1	1 0 1 1 1 0 1 0	186	0xBA	8
59	0x3B	1	1	1 1 1 1 1 0 1 1	251	0xFB	8
60[a]	0x3C	0	0	0 0 1 1 1 1 0 0	60	0x3C	8
61[b]	0x3D	1	0	0 1 1 1 1 1 0 1	125	0x7D	8
62[c]	0x3E	1	1	1 1 1 1 1 1 1 0	254	0xFE	8
63[d]	0x3F	0	1	1 0 1 1 1 1 1 1	191	0xBF	8

a. 标识符 60（0x3C）保留用于主机请求命令帧（见 3.2 节）。

b. 标识符 61（0x3D）保留用于从机响应命令帧。

c. 标识符 62（0x3E）保留用于用户定义的扩展帧（见 3.2 节）。

d. 标识符 63（0x3F）保留用于以后的 LIN 扩展格式。

A.3 校验和计算举例

假设：

报文帧有 4 个字节。

Data0 = 0x4A

Data1 = 0x55

Data2 = 0x93

Data3 = 0xE5

	hex	CY	D7	D6	D5	D4	D3	D2	D1	D0
0x4A	0x4A		0	1	0	0	1	0	1	0
+0x55=	0x9F	0	1	0	0	1	1	1	1	1
（加进位）	0x9F		1	0	0	1	1	1	1	1
+0x93=	0x132	1	0	0	1	1	0	0	1	0
（加进位）	0x33		0	0	1	1	0	0	1	1
+0xE5=	0x118	1	0	0	0	1	1	0	0	0
（加进位）	0x19		0	0	0	1	1	0	0	1
取反	0xE6		1	1	1	0	0	1	1	0
0x19+0xE6=	0xFF		1	1	1	1	1	1	1	1

得出的校验和是 0x19。校验字节是 0xE6，是校验和取反。

接收的节点可以使用相同的加法机制检查数据和校验字节的一致性。校验和＋校验字节必须等于 0xFF。

A.4 报文错误的原因

下面的错误机制可以导致报文的损坏：

接地电压的本地扰动

接收方的本地接地电压比发送方低，因此，接收节点将显性的总线电平（逻辑电平是“0”）认为是隐性（逻辑电平是“1”）或无效。输入信号的电平比显性信号电平的有效范围高。产生地电压的偏移的原因可以是：举个例子，在对地连接的寄生电阻上流过很高的负载电流。

通过发送节点监视总线电平将无法检测这个扰动。

电源电压的本地扰动

接收器的本地电源电压比发送器的高，所以，接收节点将隐性的总线电平（逻辑电平是“1”）认为是显性（逻辑电平是“0”）或无效。输入信号的电平比隐性电平的有效范围低。本地电压上升的原因是：譬如，内部电子电压的二极管—电容电压缓冲。如果网络中有电压降，电容会暂时保持接收方内部电源电压，因而比发送方内部电源电压高。

通过发送节点监视总线电平将无法检测这个扰动。

总线信号的总体电子扰动

总线上的电压可以被譬如电磁干涉等因素扰动，此时的逻辑总线值是不正确的。

通过发送节点监视总线电平将可以检测这个扰动。

不同步时基

如果从机节点的时基和主机节点的有显著的偏离，则在定义的位定时窗口中不会采样输入的数据位或发送输出的数据位（见第 9 章）。

通过发送节点监视总线电平将无法检测这个扰动。发送的从机将正确接收到自己的报文，但主机或其他从机将接收到用“错误的频率”发送的不正确报文。

A.5 故障界定的建议

特殊的故障界定并不是 LIN 协议规范的一部分。在执行故障界定时，我们建议使用下面的过程：

A.5.1 主机控制单元

- 主机任务发送：在回读自己的发送时可以检测到同步字节或标识符字节的位错误。

主机控制单元通过增加主机发送错误计数器（MasterTransmitErrorCounter）来保存任何发送错误的轨迹。当发送同步或标识符场被本地损坏时，计数器每次都加 8。当两个场回读都正确时，计数器每次都减 1（不低于 0）。

如果计数器的值超过 C_MASTER_TRANSMIT_ERROR-THRESHOLD（假设总线上有重大的扰动），应用层将执行错误处理过程。

- 在主机控制单元中的从机任务发送：

在回读自己的发送时可以检测到数据场或校验和场的位错误。

- 在主机控制单元中的从机任务接收：

当从总线上读或等待一个数据时，可以检测到从机不响应错误或校验和错误。

主机控制单元通过增加网络中每个可能的从机节点所提供的主机接收错误计数器[从机节点数量]（MasterReceiveErrorCounter）来保存任何传输错误的轨迹。当没有接收到有效的数据场或校验和场，计数器每次都加 8。当两个场都正确接收时，计数器每次都减 1（不低于 0）。

如果计数器的值超过 C_MASTER_RECEIVE_ERROR-THRESHOLD（假设连接的从机节点不正常工作），应用层将执行错误处理过程。

表 A:5.1 故障界定的错误变量

错误变量	建议的默认值
C_MASTER_TRANSMIT_ERROR-THRESHOLD	64
C_MASTER_RECEIVE_ERROR-THRESHOLD	64

A.5.2 从机控制单元

- 从机任务发送：

当回读自己的传输时可以检测到数据场和校验和场的位错误。

- 从机任务接收：

从总线上读值可检测校验和错误。如果检出校验和错误，从机将错误计数器加 8，并假设如果这是仅由特殊节点（可被主机检测到）产生的一个报文，则其他的发送节点损坏。如果所有的报文看起来都像是损坏的，则假设它自己的接收器电路有错误。如果正确接收到报文，错误计数器每次都减 1。

如果这个信息和这个控制单元的应用无关，报文的响应部分（数据和校验和场）可以不需要处理，譬如可以省略校验和计算。

如果从机在 6.1 节指定的时间内没有看到任何总线活动（NO-Bus-Activity），它将假设主机是不活动的。基于错误的处理，将启动一个唤醒过程或从机进入“limp-home”模式。

假设内部时钟远离（定义的）范围，如 3.3 节和 6.1 节所述，从机看不到任何有效的同步报文，只能看到总线的通讯。从机要重新初始化，否则不能进入 limp-home 模式。由于从机不响应任何报文，错误的处理将由主机完成。

假设主机不向从机要求任何服务，从机将暂时空闲，可以接收有效的同步报文。此时，从机可以进入 limp-home 模式。

A.6 物理接口的电源电压定义

VBAT 表示控制单元连接器的电源电压。这个单元中电气部件的内部电源电压 VSUP 和 VBAT 不同（见图 A.6.1）。它可以保护滤波器元件和总线上的动态电压变化。这在 LIN 中使用半导体元件时要考虑到。

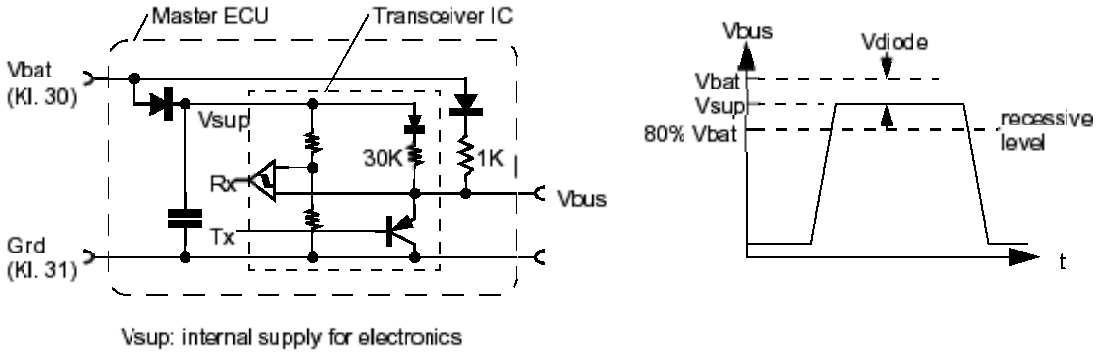


图 A.6.1 外部电源电压 VBAT 和内部电源电压 VSUP 的差异示意图

LIN 配置语言规范（V1.2）

目录

1．介绍 2
1.1 本文档的目的？ 2
2．修订历史 2
2.1 1.0 和 1.1 版的不同点 2
2.2 1.1 和 1.11 版的不同点 2
2.3 1.11 和 1.2 版的不同点 2
3．参考文献 2
4．术语 2
4.1 缩写 2
5．通用范围 3
6．句法的概述 3
7．LIN 描述文件的定义 3
7.1 LIN 协议的版本号定义 4
7.2 LIN 语言版本号的定义 4
7.3 LIN 速度的定义 4
7.4 节点定义 4
7.5 节点诊断地址的定义 4
7.6 信号定义 4
7.7 帧的定义 5
7.8 事件触发帧的定义 6
7.9 诊断帧的定义 6
7.10 进度表的定义 7
7.11 信号组的定义 8
7.12 信号编码类型定义 9
7.13 信号表示的定义 10
8．例子 10
8.1 LIN 描述文件 10

1．介绍

这个文档是 LIN 规范的一部分。

1.1 本文档的目的？

本文档讲述 LIN 描述语言的句法和语义，可由 LIN 工具识别的。

2．修订历史

修订版本	作者	日期	描述
1.0	VCT-IHt	99-07-02	规范的第一版
1.1	VCT-IHt	99-12-14	改进句法，纠正一个错误
1.11	VCT-IHt	00-02-11	删去前面的第 8 章
1.2	VCT-IHt	00-08-28	加入事件触发帧的定义
1.2	VCT-IHt	00-11-13	加入可选的帧长度定义
草稿 2			加入诊断帧处理
			改变事件触发帧的定义

2.1 1.0 和 1.1 版的不同点

- LIN 协议和语言版本号在语法中被改成 char-string。
- 纠正了组_偏移的描述错误。
- 改进了解码类型的语法。
- 根据修改升级范例。

2.2 1.1 和 1.11 版的不同点

- 删去仿真控制文件的描述
- 纠正进度表例子的错误

2.3 1.11 和 1.2 版的不同点

- 增加事件触发帧的定义选项
- 增加可选帧长度的定义
- 改变事件触发帧的定义
- 增加诊断地址的定义
- 增加诊断帧的定义

3．参考文献

[1] LIN Protocol Specification 1.2

[2] LIN API Recommended Practice 1.2

4．术语

4.1 缩写

LIN	Local Interconnect Network
TBD	To be defined（待定义）
Tool	LIN analyser/emulator（LIN 分析仪／仿真器）

5．通用范围

本文档所描述的语言是用于建立一个“LIN 描述文件”。LIN 描述文件描写了整个 LIN 网络，而且包含了监控网络所需的所有信息。通过工具的用户接口，这些信息足够可以进行有限的仿真（如果工具支持）控制。（例如：选择仿真节点，选择进度表）

LIN 工具的用户接口没有定义句法或语义，使工具供应商可以开发特殊的工具。

另外，LIN 描述文件能被单个部件引用，用于向指定 LIN 网络中的一个电子控制单元写入软件。应用程序接口（API）被定义操作规程建议（见参考资料[2]），可在不同的应用程序中用一种唯一的方法访问 LIN 网络。但，LIN 描述文件不能访问应用程序的功能特征。

6．句法的概述

下面的句法用经过修改的 BNF（Bachus-Naur Format）来描写：

符号	意义
::=	在::=左边的名字用在它右边的句法来表示
<>	用于标记后面定义的对象（Used to mark objects specified later）
|	表示选择。可以选择出现在左边或右边的内容
Bold	粗体的文字是保留的，它或者是一个保留的字或是一个强制的标点
[ ]	方括号中的文字将出现一次或几次
（ ）	将一些选择子句组合到一起
char_string	在引号中的任何字符串
identifier	标识符。通常用于对象命名；在定义变量时，标识符要符合普通 C 语言的规则
integer	整数。十进制整数（第一个数字是 1～9）或十六进制整数（前缀是 0x）
real_or_integer	实数或整数。实数通常为十进制，而且有一个小数点

在文件的任何地方都可以使用句法注释。注释的句法和 C++的一样，是从//到一行的结尾或忽略在定界符/*和*/之间的内容。

7．LIN 描述文件的定义

<LIN_description_file> ::=
LIN_description_file ;
<LIN_protocol_version_def>
<LIN_language_version_def>
<LIN_speed_def>
<Node_def>
(<Diag_addr_def>)
<Signal_def>
(<Diag_signal_def>)
<Frame_def>
(<Event_triggered_frame_def>)
(<Diag_frame_def>)
<Schedule_table_def>
(<Signal_groups_def>)
(<Signal_encoding_type_def>)
(<Signal_representation_def>)

7.1 LIN 协议的版本号定义

<LIN_protocol_version_def> ::=

LIN_protocol_version = char_string;

范围可以从“0.01”到“99.99”。

7.2 LIN 语言版本号的定义

<LIN_language_version_def> ::=

LIN_language_version = char_string ;

范围可以从“0.01”到“99.99”。

7.3 LIN 速度的定义

<LIN_speed_def> ::=

LIN_speed = real_or_integer kbps ;

范围可以从 5.00 到 20.00kbit/s。

7.4 节点定义

<Node_def >::=

Nodes {

Master:<node_name>,<time_base> ms ,<jitter> ms ;

Slaves:<node_name>([,<node_name>]) ;

}

<node_name> ::= identifier

所有的 node_name 标识符要在 Nodes 的子集中唯一。

在 Master 保留字后的 node_name 标识符定义了主机节点。

<time_base> ::= real_or_integer

time_base 的值定义了主机节点使用的时基，以产生允许的最大帧传输时间。时基可以定义在毫秒级。

<jitter> ::= real_or_integer

jitter 的值定义了从时基起始点到帧头起始点（BREAK 信号的下降沿）之间的最大和最小偏差。jitter 应定义在毫秒级。（有关 time_base 和 jitter 使用的相关信息，请参见 Schedule_tables 子集的定义。）

7.5 节点诊断地址的定义

<Diag_addr_def >::=

Diagnostic_addresses {

[<node_name>:<diag_addr>;]

}

<node_name> ::= identifier

所有的 node_name 标识符应和在 Nodes 子集中定义的其中一个 node_name 标识符相等。

<diag_addr> ::= integer

diag_addr 在范围 1～255 之间定义了节点的诊断地址。（诊断地址 0 保留到以后使用。）

7.6 信号定义

<Signal_def> ::=

Signals {

[< signal_name >:< signal_size>,<init_value>,<published_by>

[,<subscribed_by>];]

}

<signal_name> ::= identifier

所有 signal_name 标识符在 Signals 子集中应唯一。

<signal_size> ::= integer

signal_size 是在 1～16 位之间，它定义了信号的大小。

<init_value> ::= integer

init_value 定义了可以被所有用户节点使用的信号值，直到帧所包含的信号被接收。相同的初始信号值应当从发布者节点发送（根据进度表），直到应用程序升级信号。

<published_by> ::= identifier

<subscribed_by> ::= identifier

published_by 标识符和 subscribed_by 标识符要和在 Nodes 子集中定义的其中一个 node_name 标识符相等。

7.7 帧的定义

<Frame_def> ::=

Frames {

[<frame_name>:<frame_id>,<published_by>(,<frame_size>) {

[<signal_name>,<signal_offset>;]

}]

}

<frame_name> ::= identifier

所有的 frame_name 标识符应在 Frames 子集中唯一。

<frame_id> ::= integer

frame_id 的范围是 0～63，它定义了帧的 ID 号。这个 ID 号在 Frames 子集的所有帧中唯一。

根据 LIN 协议规范，帧的大小继承帧的 ID。

<published_by> ::= identifier

published_by 标识符要和在 Nodes 子集中定义的其中一个 node_name 标识符相等。

<frame_size> ::= integer

frame_size 是可选项，范围是 0～8，定义了帧的大小。如果 frame_size 的规范不存在，则如在 LIN 协议规范中定义的，帧的大小将偏离 frame_id。

<signal_name> ::= identifier

signal_name 标识符应和在 Signals 子集中定义的其中一个 signal_name 标识符相等。

在一个帧中定义所有信号应被由 published_by 标识符定义的相同的节点发布。

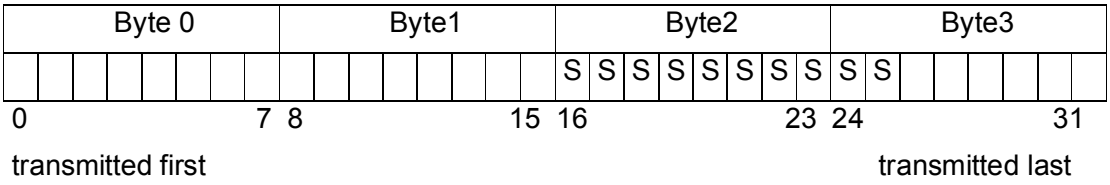
<signal_offset> ::= integer

signal_offset 的值定义了帧中信号最低位的位置。它的值可以从 0 到（8*frame_size-1）。信号的最低位将被首先发送。

举例：

信号“S”（大小＝10bits）被放置于一个 4 字节的帧中，偏移是 16。

（S 的 LSB 在偏移 16，MSB 在偏移 25。）



信号的封装规则：

- 强迫所有 8 位或大于 8 位信号的（最低位）字节对齐。
- 小于 8 位的信号要求包含在一个单个字节中（因此，字节定界不能穿过“小”信号）。

7.8 事件触发帧的定义

<Event_triggered_frame_def> ::=

Event_triggered_frames {

[<event_trig_frm_name>:<frame_id>[,<frame_name>];]

}

<event_trig_frm_name> ::= identifier

所有 event_trig_frm_name 标识符应在 Event_triggered_frames 的子集中唯一，而且和在 Frames 子集中定义的所有标识符都不同。

<frame_id> ::= integer

frame_id 的范围是 0～63，它定义了帧的 ID 号。这个 ID 号在 Frames 子集和 Event_triggered_frames 子集的所有帧中唯一。

<frame_name> ::= identifier

frame_name 标识符应等于 Frames 子集中定义的其中一个标识符。

由 frame_name 标识符列表定义的所有帧可以通过网络上不同的节点发布。

注意：当由 frame_name 在 Frames 子集中定义一个帧并映射到一个事件触发帧时，这个定义的帧在设计中将受到额外的限制。一个事件触发帧需要使用保留的第一字节。这个字节包括在 Frames 子集中定义的完整的帧 id（标识符和奇偶校验位）。

只有当帧的内容在上次事件触发帧的传输后升级了，从机任务才会响应事件触发帧的 id（在 Event_triggered_frames 子集中定义）。

注意：当事件触发帧和在 Frames 子集中定义的帧在网络中同时有效时，这两个帧的数据内容应一致。如果有多于一个节点同时响应事件触发帧，则会产生总线冲突。在这种情况下，主机 ECU 负责查询各个从机的 ECU 以获得在 Frames 子集中定义的带帧 id 的所有映射帧。

7.9 诊断帧的定义

<Diag_frame_def> ::=

Diagnostic_frames {

MasterReq : 60 {

MasterReqB0,0;

MasterReqB1,8;

MasterReqB2,16;

MasterReqB3,24;

MasterReqB4,32;

```
            MasterReqB5,40;
            MasterReqB6,48;
            MasterReqB7,56;
    }
    SlaveResp : 61 {
            SlaveRespB0,0;
            SlaveRespB1,8;
            SlaveRespB2,16;
            SlaveRespB3,24;
            SlaveRespB4,32;
            SlaveRespB5,40;
            SlaveRespB6,48;
            SlaveRespB7,56;
      }
}
```

保留帧 MasterReq 和 SlaveResp 的名字用于识别诊断帧。

MasterReq 有在 LIN 协议规范中定义的固定的帧 ID（60）和固定的长度（8 字节）。MasterReq 只能由主机节点发送。

SlaveResp 有在 LIN 协议规范中定义的固定的帧 ID（61）和固定的长度（8 字节）。SlaveResp 只能由前面的 MasterReq 帧所选择的从机节点发送。从机节点的选择是由在 Diagnostic_addresses 子集中定义的从机的诊断地址决定。

保留的信号名字 MasterReqB0 到 MasterReqB7 将 MasterReq 帧中的信号定义成 8 位的长整数。

保留的信号名字 SlaveRespB0 到 SlaveRespB7 将 SlaveResp 帧中的信号定义成 8 位的长整数。

MasterReqB0 和 SlaveRespB0 可以如 LIN 协议规范 3.2 节所定义的进行使用。

预定义诊断信号（带有 MasterReq 和 SlaveResp 帧）的封装描述在普通的帧描述句法后面。

(signal_name,signal_offset;)

7.10 进度表的定义

```
<Schedule_table_def> ::=
Schedule_tables {
    [<schedule_table_name> {
        [<frame_name> delay <frame_time> ms ;]
    }]
}
```

<schedule_table_name> ::= identifier

所有的 schedule_table_name 标识符应在 Schedule tables 子集中唯一。

<frame_name> ::= identifier

frame_name 标识符应等于在 Frames 子集中定义的 frame_name 标识符的其中一个。

<frame_time> ::= real_or_integer

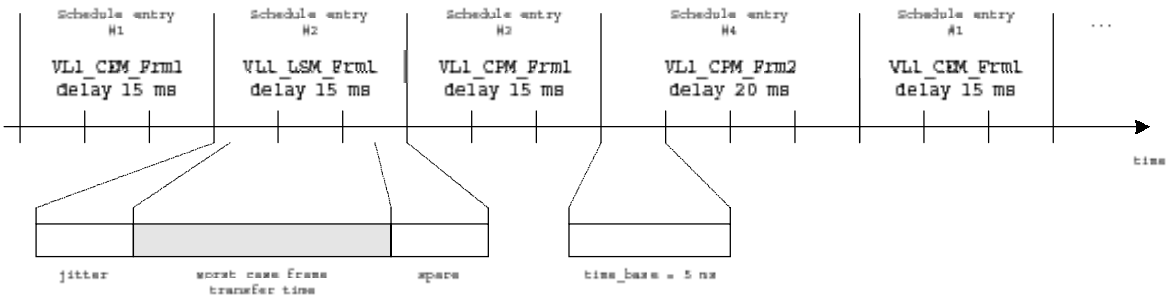
frame_time 定义了两个相邻帧之间的时间间隔。这个时间要比允许帧传输的最大时间长，而且应当是主机节点时基值的精确倍数。frame_time 的值应定义在毫秒级。

进度表的选择由主机的应用程序控制。

进度表之间的切换要在帧时间（当前发送的帧）过去后立即完成。

举例：

```
Schedule_tables {
    VL1_ST1 {
        VL1_CEM_Frm1 delay 15 ms;
        VL1_LSM_Frm1 delay 15 ms;
        VL1_CPM_Frm1 delay 15 ms;
        VL1_CPM_Frm2 delay 20 ms;
    }
}
```



每个进度表表项的延时要比 jitter 和最坏情况下的帧传输时间长（见参考文献[1]）。

注意！如何使用以及在不同的进度表中切换是应用程序的问题，但相应的机制在参考文献[2]中有描述。

7.11 信号组的定义

信号组子集在 LIN 文件中是一个可选项。

```
<Signal_groups_def> ::=
Signal_groups {
  [<signal_group_name>:<group_size> {
      [<signal_name>,<group_offset> ;]
  }]
}
```

<signal_group_name> ::= identifier

signal_group_name 标识符应在 signal_group 子集中唯一，而且和在 Signals 子集中定义的 signal_name 标识符不同。

<group_size> ::= integer

group_size 的范围是 1～（8*frame_size）位，它定义了信号的长度。

<signal_name> ::= identifier

signal_name 标识符应和 Signals 子集的其中一个 signal_name 标识符相同。

<group_offset> ::= integer

group_offset 的值定义了组中信号最低位的位置。它值的范围是 0～（group_size-1）。最低位被首先发送。组中不使用的位（位置）应填上 0。

信号组通常由一个帧中的信号组成。根据组的定义，大于 16 位的单个信号可以用工具显示（使用预定义的信号编码类型）。

7.12 信号编码类型定义

信号编码类型子集是 LIN 文件的可选部分。

```
<signal_encoding_type_def> ::=
Signal_encoding_types {
    [<signal_encoding_type_name> {
        [<logical_value> |
        <physical_range> |
        <bcd_range> |
        <ascii_range>]
    }]
}
<signal_encoding_type_name> ::= identifier
```

所有的 signal_encoding_type_name 标识符应在 Signal encoding types 子集中唯一。

```
<logical_value> ::= logical_value,<signal_value>(,<text_info>) ;
<physical_range> ::= physical_value,<min_value>,<max_value>,<scale>,
<offset>(,<text_info>) ;
<bcd_value> ::= bcd_value ;
<ascii_value> ::= ascii_value ;
<signal_value> ::= integer
<min_value> ::= integer
<max_value> ::= integer
<scale> ::= real_or_integer
<offset> ::= real_or_integer
<text_info> ::= char_string
```

其中 signal_value、min_value、max_value 的值范围在 0～65535 之间。

max_value 的值要大于或等于 min_value。信号编码类型信息可以由工具使用，用逻辑 / 放大的物理值和 / 或监控过程中一个预定义的字符串来代替原始值。如果原始值在定义的最大和最小值范围内，物理值可以用下面的算式计算：

物理值＝放大倍数*原始值＋偏移

举例：

```
Signal_encoding_types {
    1BitDig {
        logical_value,0,”off”;
        logical_value,1,”on”;
    }
    Temp {
        physical_value,0,250,0.5,-40,”degree”;
        physical_value,251,253,1,0,”undefined”;
        logical_value,254,”out of range”;
        logical_value,255,”error”;
    }
}
```

7.13 信号表示的定义

信号表示子集是 LIN 文件中的可选项。

<Signal_representation_def> ::=

Signal_representation {

[<signal_encoding_type_name>:<signal_or_group_name>

([,<signal_or_group_name>]);]

}

<signal_encoding_type_name> ::= identifier

signal_encoding_type_name 标识符应和在 signal_encoding_types 子集中定义的其中一个 signal_encoding_type_name 标识符相同。

<signal_or_group_name> ::= <signal_name> | <signal_group_name>

<signal_name> ::= identifier

<signal_group_name> ::= identifier

signal_name 标识符应是和 Signals 子集中定义的其中一个 signal_name 标识符相等。

signal_group_name 标识符应是和 signal_groups 子集中定义的其中一个 signal_group_name 标识符相等。

8. 例子

8.1 LIN 描述文件

```
// 这是一个 LIN 描述范例文件
// 由 Istvan Horvath 发布
LIN_description_file ;
LIN_protocol_version = “1.0”;
LIN_language_version = “1.1”;
LIN_speed = 19.2 kbps;

Nodes {
    Master:CEM,5 ms, 0.1 ms;
    Slaves:LSM,CPM;
}

Signals {
    RearFogLampInd:1,0,CEM,LSM;
    PositionLampInd:1,0,CEM,LSM;
    FrontFogLampInd:1,0,CEM,LSM;
    IgnitionKeyPos:3,0,CEM,LSM,CPM;
    LSMFuncIllum:4,0,CEM,LSM;
    LSMSymbolIllum:4,0,CEM;
    StartHeater:3,0,CEM;
    CPMReqB0:8,0,CEM;
    CPMReqB1:8,0,CEM;
    CPMReqB2:8,0,CEM;
    CPMReqB3:8,0,CEM;
    CPMReqB4:8,0,CEM;
```

```
        CPMReqB5:8,0,CEM;
        CPMReqB6:8,0,CEM;
        CPMReqB7:8,0,CEM;
        ReostatPos:4,0,LSM;
        HeadLampBeamLev:4,0,LSM;
        FrontFogLampSw:1,0,LSM;
        RearFogLampSw:1,0,LSM;
        MLSOff:1,0,LSM;
        MLSHeadLight:1,0,LSM;
        MLSPosLight:1,0,LSM;
        HBLSortHigh:1,0,LSM;
        HBLShortLow:1,0,LSM;
        ReoShortHigh:1,0,LSM;
        ReoShortLow:1,0,LSM;
        LSMHWPartNoB0:8,0,LSM;
        LSMHWPartNoB1:8,0,LSM;
        LSMHWPartNoB2:8,0,LSM;
        LSMHWPartNoB3:8,0,LSM;
        LSMSWPartNo:8,0,LSM;
        CPMOutputs:10,0,CPM;
        HeaterStatus:4,0,CPM;
        CPMGlowPlug:7,0,CPM;
        CPMFanPWM:8,0,CPM;
        WaterTempLow:8,0,CPM;
        WaterTempHigh:8,0,CPM;
        CPMFuelPump:7,0,CPM;
        CPMRunTime:13,0,CPM;
        FanIdealSpeed:8,0,CPM;
        FanMeasSpeed:8,0,CPM;
        CPMRespB0:1,0,CPM;
        CPMRespB1:1,0,CPM;
        CPMRespB2:1,0,CPM;
        CPMRespB3:1,0,CPM;
        CPMRespB4:1,0,CPM;
        CPMRespB5:1,0,CPM;
        CPMRespB6:1,0,CPM;
        CPMRespB7:1,0,CPM;
    }

    Frames {
        VL1_CEM_Frm1:32,CEM {
                RearFogLampInd,0;
                PositionLampInd,1;
                FrontFogLampInd,2;
```

```
        IgnitionKeyPos,3;
        LSMFuncIllum,8;
        LSMSymbolIllum,12;
        StartHeater,16;
}

VL1_CEM_Frm2:48,CEM {
        CPMReqB0,0;
        CPMReqB1,8;
        CPMReqB2,16;
        CPMReqB3,24;
        CPMReqB4,32;
        CPMReqB5,40;
        CPMReqB6,48;
        CPMReqB7,56;
}

VL1_LSM_Frm1:33,LSM {
        ReostatPos,0;
        HeadLampBeamLev,4;
        FrontFogLampSw,8;
        RearFogLampSw,9;
        MLSOff,10;
        MLSHeadLight,11;
        MLSPosLight,12;
        HBLSortHigh,16;
        HBLShortLow,17;
        ReoShortHigh,18;
        ReoShortLow,19;
}

VL1_LSM_Frm2:49,LSM {
        LSMHWPartNoB0,0;
        LSMHWPartNoB1,8;
        LSMHWPartNoB2,16;
        LSMHWPartNoB3,32;
        LSMSWPartNo,40;
}

VL1_CPM_Frm1:50,CPM {
        CPMOutputs,0;
        HeaterStatus,10;
        CPMGlowPlug,16;
        CPMFanPWM,24;
```

```
            WaterTempLow,32;
            WaterTempHigh,40;
            CPMFuelPump,56;
        }

        VL1_CPM_Frm2:34,CPM {
            CPMRunTime,0;
            FanIdealSpeed,16;
            FanMeasSpeed,24;
        }

        VL1_CPM_Frm3:51,CPM {
            CPMRespB0,0;
            CPMRespB1,8;
            CPMRespB2,16;
            CPMRespB3,24;
            CPMRespB4,32;
            CPMRespB5,40;
            CPMRespB6,48;
            CPMRespB7,56;
        }
    }
    Schedule_tables {
        VL1_ST1 {
            VL1_CEM_Frm1 delay 15 ms;
            VL1_LSM_Frm1 delay 15 ms;
            VL1_CPM_Frm1 delay 20 ms;
            VL1_CPM_Frm2 delay 20 ms;
        }

        VL1_ST2 {
            VL1_CEM_Frm1 delay 15 ms;
            VL1_CEM_Frm2 delay 20 ms;
            VL1_LSM_Frm1 delay 15 ms;
            VL1_LSM_Frm2 delay 20 ms;
            VL1_CEM_Frm1 delay 15 ms;
            VL1_CPM_Frm1 delay 20 ms;
            VL1_CPM_Frm2 delay 20 ms;
            VL1_LSM_Frm1 delay 15 ms;
            VL1_CPM_Frm3 delay 20 ms;
        }
    }

    Signal_groups {
```

```
        CPMReq:64 {
                CPMReqB0,0;
                CPMReqB1,8;
                CPMReqB2,16;
                CPMReqB3,24;
                CPMReqB4,32;
                CPMReqB5,40;
                CPMReqB6,48;
                CPMReqB7,56;
        }
    }

    Signal_encoding_types {
        1BitDig {
                logical_value,0,”off”;
                logical_value,1,”on”;
        }
        2BitDig {
                logical_value,0,”off”;
                logical_value,1,”on”;
                logical_value,2,”error”;
                logical_value,3,”void”;
        }
        Temp {
                physical_value,0,250,0.5,-40,”degree”;
                physical_value,251,253,1,0,”undefined”;
                logical_value,254,”out of range”;
                logical_value,255,”error”;
        }
        Speed {
                physical_value,0,65500,0.008,250,”km/h”;
                physical_value,65501,65533,1,0,”undefined”;
                logical_value,65534,”error”;
                logical_value,65535,”void”;
        }
    }

    Signal_representations {
        1BitDig:RearFogLampInd,PositionLampInd,FrontFogLampInd;
        Temp:WaterTempLow,WaterTempHigh;
        Speed:FanIdealSpeed,FanMeasSpeed;
    }
```

LIN API 操作规程建议（V1.2）

目录

1．介绍 2
1.1 本文档的目的 2
2．修订历史 2
3．参考文献 2
4．术语 2
4.1 缩写 2
5．总体描述 2
6．API 规范 3
6.1 初始化 3
6.1.1 l_sys_init 3
6.2 信号调用 4
6.2.1 信号类型 4
6.2.2 读调用 4
6.2.3 写调用 4
6.3 标志调用 4
6.3.1 l_flg_tst 5
6.3.2 l_flg_clr 5
6.4 过程调用 5
6.4.1 l_sch_tick 5
6.4.2 l_sch_set 6
6.5 接口调用 6
6.5.1 l_ifc_init 6
6.5.2 l_ifc_connect 6
6.5.3 l_ifc_disconnect 7
6.5.4 l_ifc_ioctl 7
6.5.5 l_ifc_rx 7
6.5.6 l_ifc_tx 7
6.5.7 l_ifc_aux 8
6.6 用户提供的调用 8
6.6.1 l_sys_irq_disable 8
6.6.2 l_sys_irq_restore 8
7．范例 9
7.1 LIN API 的使用 9
7.2 LIN 描述文件 11

1．介绍

本文档是有关在 LIN SW 模块中使用应用程序接口（API）的操作规程建议，并作为对 LIN 标准规范的补充。

1.1 本文档的目的

本文档的目的是为 LIN SW 模块和 SW 应用定义一个合适的接口。

2．修订历史

修订版本	作者	日期	描述
1.0	VCT-CBn	00-02-10	操作规程建议的第一版
1.1	VCT-CBn	00-08-28	根据 00-07-03 的改变请求升级。 改变 l_sch_tick 函数的返回值政策，在 l_sch_set 函数中增加输入参数。
1.2	VCT-CBn	00-11-13	根据 00-09-26 的改变请求升级。 在 l_ifc_connect 和 l_ifc_disconnect 函数中增加返回值。 在 l_ifc_init 函数中加入额外的描述文字。 纠正动态 l_ifc_ioctl 函数的返回值类型错误。

3．参考文献

[1] LIN Configuration language specification 1.11

4．术语

4.1 缩写

API	Application Program Interface（应用程序接口）
ECU	Electronic Control Unit (with μ-Controller/μ-Processor)（带微控制器/微处理器的电子控制单元）
LCFG	LIN Configuration tool (PC-program)（LIN 配置工具）
LIN	Local Interconnect Network
TBD	To be defined（待定义）

5．总体描述

本章叙介绍了如何在应用程序中集成 LIN API 的一个可行的解决方案。

LIN API 是一个网络软件层，它在用户为任意的 ECU 建立应用程序时，隐藏了 LIN 网络配置的详细情况（例如：信号如何映射到相应的帧）。所以，它提供了一个 API 给用户，这个 API 着重于 LIN 网络的信号传输。前面介绍的 PC 工具“LCFG”将处理从网络配置到完成程序代码的这些步骤。它为用户提供了配置灵活性。

LCFG 的输入是一个或很多 LIN 网络配置文件（见参考文献[1]）和一个本地描述文件（另外叙述）。网络配置文件包括这个专用 LIN 网络的完整定义。本地文件叙述节点的专用实体（譬如：连接到信号 / 帧和 HW 规范的标志）。

LCFG 可以产生 ANSI C 文件和 H 文件，这些文件应当和应用 SW 一起编译。而且用户可将提供的 LIN 库包含到应用程序中。使用 LIN 库和 API 来建立用 LIN 作为通讯接口的应用的工作流程图如下图 1 所示。

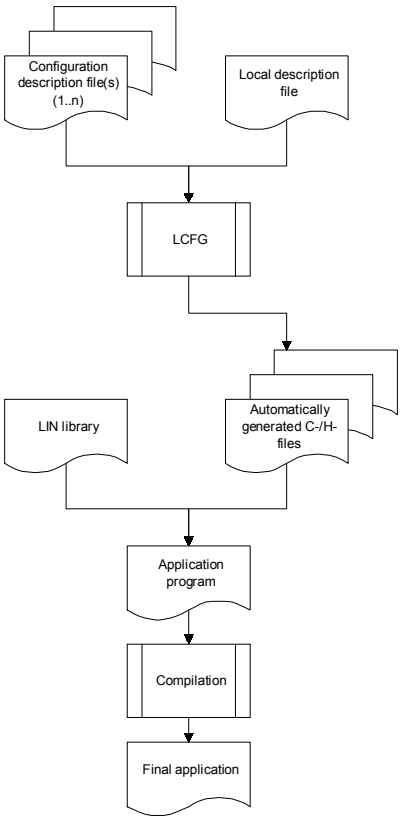


图 1 LIN 配置的工作流程图

6．API 规范

LIN API 是一组函数，它们为了减少和已有的 SW 冲突的风险而给 API 一个独立的名字空间。所有函数和类型都有前缀“l_”（小写的“L”和一个“下划线”）。

LIN SW 定义了以下的类型：

- l_bool
- l_ioctl_op
- l_irqmask
- l_u8
- l_u16

为了获得效率，这些函数大多数是静态函数（C 预处理器宏#define，由 LIN 配置工具自动产生）。

6.1 初始化

6.1.1 l_sys_init

原型

l_bool l_sys_init(void);

描述

l_sys_init 执行 LIN SW 的初始化。

返回值

“0”：如果成功初始化。

“非零”：初始化失败。

注意：

l_sys_init 的调用是用户在 LIN SW 中使用其他任何 API 函数前必须使用的第一个调用。

6.2 信号调用

6.2.1 信号类型

信号有 3 种不同的类型：

- l_bool: 一位的信号（如果出错则是 0，其他情况是非零）
- l_u8: 1～8 位大小的信号
- l_u16: 9～16 位大小的信号

6.2.2 读调用

动态原型

```
l_bool l_bool_rd(l_signal_handle sss);
l_u8 l_u8_rd(l_signal_handle sss);
l_u16 l_u16_rd(l_signal_handle sss);
```

静态应用

```
l_bool l_bool_rd_sss(void);
l_u8 l_u8_rd_sss(void);
l_u16 l_u16_rd_sss(void);
```

其中 sss 是信号的名字（例如：l_u8_rd_EngineSpeed()）

描述

读和返回名字是 sss 的信号的当前值。

注意

无

6.2.3 写调用

动态原型

```
void l_bool_wr(l_signal_handle sss, l_bool v);
void l_u8_wr(l_signal_handle sss, l_u8 v);
void l_u16_wr(l_signal_handle sss, l_u16 v);
```

静态应用

```
void l_bool_wr_sss(l_bool v);
void l_u8_wr_sss(l_u8 v);
void l_u16_wr_sss(l_u16 v);
```

其中 sss 是信号的名字（例如：l_u8_wr_EngineSpeed(v)）。

描述

将名字是 sss 的信号的当前值设置成 v。

注意

无

6.3 标志调用

标志是 ECU 的本地对象，用于应用和 LIN SW 之间的同步。标志可以由 LIN SW 自动设置，应用程序只能测试 / 清除标志。

6.3.1 l_flg_tst

动态原型

```
l_bool l_flg_tst(l_flag_handle fff);
```

静态应用

```
l_bool l_flg_tst_fff(void);
```

其中 fff 是标志的名字（例如：l_flg_tst_RxEngineSpeed()）。

描述

返回值是 C 的布尔类型，表示由名字 fff 指定的标志的当前状态（即：如果标志被清除则返回 0，否则返回值非零）。

注意

无

6.3.2 l_flg_clr

动态原型

```
void l_flg_clr(l_flag_handle fff);
```

静态应用

```
void l_flg_clr_fff(void);
```

其中 fff 是标志的名字（例如：l_flg_clr_RxEngineSpeed()）。

描述

将名字是 fff 的标志的当前值置零。

注意

无

6.4 过程调用

6.4.1 l_sch_tick

动态原型

```
l_u8 l_sch_tick(l_ifc_handle iii);
```

静态应用

```
l_u8 l_sch_tick_iii(void);
```

其中 iii 是接口的名字（例如：l_sch_tick_MyLinIfc()）。

描述

l_sch_tick 函数跟在进度表的后面。当帧变得合适时，它将被初始化发送。如果当前的进度表已经到达末尾，l_sch_tick 再次在最后一次调用 l_sch_set 时设置的最后一个进度表的开始处启动。

l_sch_tick 在 ECU 的每个接口被独立调用，其速率在网络配置文件中定义。

返回值

“非零”：如果下一个 l_sch_tick 的调用会启动下一个进度表项中帧的发送，返回值非零。返回值在这种情况下是下一个进度表项的号码（从进度表的开始算起）。所以，如果进度表有 N 个表项，则返回值可以是 1～N。

否则是“零”。

注意

l_sch_tick 只能在主机节点使用。

l_sch_tick 的调用不仅仅启动下一个帧的发送，它还会更新由前面一次 l_sch_tick 调用所接收到的信号值（即，这个接口的上一个帧）。

（见 l_sch_set 返回值的使用注意事项。）

6.4.2 l_sch_set

动态原型

```
void l_sch_set(l_ifc_handle iii, l_schedule_handle sch, l_u8 ent);
```

静态应用

```
void l_sch_set_iii(l_schedule_handle sch, l_u8 ent);
```

其中 iii 是接口的名字（譬如：l_sch_set_MyLinIfc(MySchedule1,0)）。

描述

设置下一个进度表 sch 紧跟着 l_sch_tick 函数，接口是 iii。在当前的进度表到达它的下一个进度表的表项时，新的进度表被立即激活。

输入参数 ent 定义了起始的新进度表的起始表项点。如果进度表有 N 个表项，ent 的值可以是 0～N，如果 ent 是 0 或 1，新的进度表将在开始的地方启动。

注意

l_sch_set 只能在主机节点使用。

Ent 的输入值和 l_sch_tick 的返回值结合，可以用于譬如：用另外一个进度表暂时中断前一个进度表，而且能返回被中断进度表的中断点。

6.5 接口调用

6.5.1 l_ifc_init

动态原型

```
void l_ifc_init(l_ifc_handle iii);
```

静态应用

```
void l_ifc_init_iii(void);
```

其中 iii 是接口的名字（例如：l_ifc_init_MyLinIfc()）。

描述

l_ifc_init 初始化名字为 iii 的控制器（例如：设置如波特率等的间隔）。由 l_ifc_init 调用的设置的默认进度表是一个空的进度表 NULL_Schedule，没有帧会被发送和接收。

注意

接口在本地描述文件中列出。

函数 l_ifc_init()的调用是用户在使用其他相关的 LIN API 函数接口（例如：l_ifc_connect()或 l_ifc_rx()）前要首先调用的一个函数。

6.5.2 l_ifc_connect

动态原型

```
l_bool l_ifc_connect(l_ifc_handle iii);
```

静态应用

```
l_bool l_ifc_connect_iii(void);
```

其中 iii 是接口的名字（譬如：l_ifc_connect_MyLinIfc()）。

描述

l_ifc_connect 的调用会将接口 iii 连接到 LIN 网络上，可以向总线传输报文头和数据。

返回值

“0”：如果“连接操作”成功

“非零”：如果“连接操作”失败

注意

无

6.5.3 l_ifc_disconnect

动态原型

```
l_bool l_ifc_disconnect(l_ifc_handle iii);
```

静态应用

```
l_bool l_ifc_disconnect_iii(void);
```

其中 iii 是接口的名字（譬如：l_ifc_disconnect_MyLinIfc()）。

描述

调用 l_ifc_disconnect 将接口 iii 和 LIN 网络断开连接，不能向总线传输报头和数据。

返回值

“0”：如果“断开操作”成功

“非零”：如果“断开操作”失败

注意

无

6.5.4 l_ifc_ioctl

动态原型

```
l_u16 l_ifc_ioctl(l_ifc_handle iii, l_ioctl_op op, void *pv);
```

静态应用

```
l_u16 l_ifc_ioctl_iii(l_ioctl_op op, void *pv);
```

其中 iii 是接口的名字（譬如：l_ifc_ioctl_MyLinIfc(MyOp,&MyPars)）。

描述

这个函数控制了协议和接口专用的参数。其中 iii 是接口的名字，在 op 中定义的操作要应用到这个接口。指针 pv 指向可选的参数块。

它所支持的操作由接口类型决定，程序员必须参考目标绑定文件中专用接口的文档。文档会指出它执行什么操作，返回值是多少。

参数块的解释基于所选的操作。一些操作不需要这个模块。这种情况下，指针 pv 可置为 NULL。当参数块是由它的接口决定相关格式时，必须考虑将文件绑定于接口规范目标。

6.5.5 l_ifc_rx

动态原型

```
void l_ifc_rx(l_ifc_handle iii);
```

静态应用

```
void l_ifc_rx_iii(void);
```

其中 iii 是接口的名字（例如：l_ifc_rx_MyLinIfc()）。

描述

当接口 iii 接收到数据的一个字符时调用。

譬如：在当 UART 接收到数据的一个字符，而进入用户定义的中断处理程序时调用。这种情况下函数必须在 UART 的控制寄存器中执行必要的操作。

注意

应用程序负责绑定中断和设置正确的接口处理（如果使用了中断）。

6.5.6 l_ifc_tx

动态原型

```
void l_ifc_tx(l_ifc_handle iii);
```

静态应用

void l_ifc_tx_iii(void);

其中 iii 是接口的名字（譬如：l_ifc_tx_MyLinIfc()）。

描述

当接口 iii 发送了数据的一个字符时调用。

例如：在当 UART 发送了数据的一个字符，而进入用户定义的中断处理程序时调用。这种情况下函数必须在 UART 的控制寄存器中执行必要的操作。

注意

应用程序负责绑定中断和设置正确的接口处理（如果使用了中断）。

这个函数在发送和 l_ifc_rx 函数调用联合的情况下可以为空。这将在目标绑定文件中为用户说明。

6.5.7 l_ifc_aux

动态原型

void l_ifc_aux(l_ifc_handle iii);

静态应用

void l_ifc_aux_iii(void);

其中 iii 是接口的名字（例如：l_ifc_aux_MyLinIfc()）。

描述

这个函数可以在从机节点上使用，用于同步主机在 iii 接口发送的 BREAK 和 SYNC 字符。

譬如：用户定义的中断处理程序调用在连接到 iii 接口的 HW 引脚的侧面检测。

注意

l_ifc_aux 只能在从机节点中使用。

这个函数与 HW 紧密衔接，实际的执行和使用在目标绑定文件中描述。

当 BREAK/SYNC 检测在 l_ifc_rx 函数中进行时，这个函数可以是空。

6.6 用户提供的调用

用户必须提供一对可以在 LIN SW 中调用的函数，它可以某些内部操作前禁能所有的控制器中断，并可以恢复进行这些操作前的状态（这些函数用于例如 l_sch_tick 函数中）。

6.6.1 l_sys_irq_disable

动态原型

l_irqmask l_sys_irq_disable(void);

描述

这个函数必须在不产生控制器中断的情况下使用。

注意

无

6.6.2 l_sys_irq_restore

动态原型

void l_sys_irq_restore(l_irqmask previous);

描述

用户使用这个函数时必须恢复 previous 标记的状态。

注意

无

7．范例

本章将给出一个非常简单的范例显示如何使用 API。这里将给出 C 应用程序代码和 LIN 描述文件。

7.1 LIN API 的使用

```
/* ************************************************************************************************************** */
/*              File:       hello.c                                                                               */
/*            Author:       Christian Bondesson                                                                   */
/*       Description:       Example code for using the LIN API in a LIN master ECU                                */
/*                          NOTE! This is using the static API!!!                                                 */
/*
**      $Header$
*/
/*      Date:       Author:         Description:                                                                  */
/*      -----        -------         ------------                                                                 */
/*    990830       VCT-CBn         * new created                                                                 */
/*    000828       VCT-CBn         * adopted to API version 1.1 (the l_sch_tick and                              */
/*                                   l_sch_set functions updated)                                                 */
/*    001113       VCT-CBn         * adopted to API version 1.2 (the l_ifc_connect                               */
/*                                   function updated with return value)                                          */
/*   include files:   */
/*   --------------   */
/*   #include file   */
#include "lin.h"
/* ************************************************************************************************************** */
/*             PROCEDURE : l_sys_irq_restore                                                                      */
/*           DESCRIPTION : Restores the interrupt mask to the one before the call to                              */
/*                         l_sys_irq_disable was made                                                             */
/*                      IN : previous - the old interrupt level                                                   */
/* ************************************************************************************************************** */
void l_sys_irq_restore(l_imask previous)
{
    /* some controller specific things...                                                                         */
} /* end l_sys_irq_restore */
/* ************************************************************************************************************** */
/*           PROCEDURE : l_sys_irq_disable                                                                        */
/*         DESCRIPTION : Disable all interrupts of the controller and returns the                                 */
/*                       interrupt level to be able to restore it later                                           */
/* ************************************************************************************************************** */
l_imask l_sys_irq_disable(void)
{
    /* some controller specific things...                                                                         */
} /* end l_sys_irq_disable */
/* ************************************************************************************************************** */
/*             INTERRUPT : lin_char_rx_handler                                                                    */
```

```
/*         DESCRIPTION : LIN recieve character interrupt handler for the interface                */
/*                              named LIN_ifc                                                      */
/* ********************************************************************************************** */
void INTERRUPT lin_char_rx_handler(void)
{
                                                    /* just call the LIN API provided func-  */
                                                    /* tion to do the actual work            */
    l_ifc_rx_MyLinIfc();
} /* end lin_char_rx_handler */


/* ********************************************************************************************** */
/*         INTERRUPT : lin_char_tx_handler                                                        */
/*      DESCRIPTION : LIN transmit character interrupt handler for the interface                  */
/*                              named LIN_ifc                                                     */
/* ********************************************************************************************** */
void INTERRUPT lin_char_tx_handler(void)
{
                                                /* just call the LIN API provided func-      */
                                                /* tion to do the actual work                */
    l_ifc_tx_MyLinIfc();
} /* end lin_char_tx_handler */
/* ********************************************************************************************** */
/*        PROCEDURE : main                                                                        */
/*      DESCRIPTION : Main program... initialisation part                                         */
/* ********************************************************************************************** */
void main(void)
{
                                                /* initialise the LIN interface              */
    if (l_sys_init())
    {
                                                /* the init of the LIN software failed       */
    }
    else
    {
        l_ifc_init_MyLinIfc();                  /* initialise the interface                  */
        if (l_ifc_connect_MyLinIfc())
        {
                                                /* connection of the LIN interface fai-      */
                                                /* led                                       */
        }
        else
        {
                                                /* connected, now ready to send/receive      */
```

```
                                                    /* set the normal schedule to run from      */
                                                    /* beginning for this specific inter-       */
                                                    /* face                                     */
            l_sch_set_MyLinIfc(MySchedule1, 0);
        }
    }
    start_main_application();                       /* ready with init, start actaul appl.      */
} /* end main */
/* 10ms based on the minimum LIN tick time, in LIN description file...                          */
void main_application_10ms(void)
{
    /* do some application specific stuff...                                                    */
                                                    /* just a small example of signal rea-      */
                                                    /* ding and writing                         */
    if (l_flg_tst_RxInternalLightsSwitch())
    {
        l_u8_wr_InternalLightsRequest(l_u8_rd_InternalLightsSwitch());
        l_flg_clr_RxInternalLightsSwitch();
    }
                                                    /* in-/output of signals, do not care       */
                                                    /* about the return value, as we will       */
                                                    /* never switch schedule anyway...          */
    (void)l_sch_tick_MyLinIfc();
} /* end main_application_10ms */
```

7.2 LIN 描述文件

```
/* ********************************************************************************************************** */
/*          File: hello.ldf                                                                                   */
/*     Author: Christian Bondesson                                                                            */
/* Description: The LIN description file for the example program                                              */
/*
**    $Header$
*/
/*    Date:          Author:          Description:                                                            */
/*    -----           -------         ------------                                                            */
/*  990830       VCT-CBn          * new created                                                               */
LIN_description_file ;
LIN_protocol_version = “1.0”;
LIN_language_version = “1.1”;
LIN_speed = 19.2 kbps;
Nodes {
    Master: CEM, 5 ms, 0.1 ms;
    Slaves: LSM;
}
```

```
Signals {
    InternalLightsRequest: 2, 0, CEM, LSM;
    InternalLightsSwitch: 2, 0, LSM, CEM;
}
Frames {
    VL1_CEM_Frm1: 1, CEM {
        InternalLightsRequest, 0;
    }
    VL1_LSM_Frm1: 2, LSM {
        InternalLightsSwitch, 0;
    }
}
Schedule_tables {
    MySchedule1 {
        VL1_CEM_Frm1 delay 15 ms;
        VL1_LSM_Frm1 delay 15 ms;
    }
}
Signal_encoding_types {
    2BitDig {
        logical_value, 0, “off”;
        logical_value, 1, “on”;
        logical_value, 2, “error”;
        logical_value, 3, “void”;
    }
}
Signal_representations {
    2BitDig: InternalLightsRequest, InternalLightsSwitch;
}
```