工学硕士学位论文

CANopen 现场总线应用层协议主站的开发与实现

宋　威

北京工业大学

2008 年 5 月

分类号： TP336　　　　单位代码： 10005
学　号： S200502115
密　级： 公开

北京工业大学硕士学位论文

题　目　**CANOPEN 现场总线应用层协议主站的开发与实现**

英文并列题目　**THE DESIGN AND IMPLEMENTATION OF A MASTER OF THE APPLICATION LAYER FOR CANOPEN**

研究生姓名： 宋　威
专　业： 检测技术与自动化装置　　研究方向： 现场总线技术与嵌入式系统应用
导师姓名： 方穗明　　职　称： 副教授

论文报告提交日期　2008 年 5 月　　学位授予日期
授予单位名称和地址　北京工业大学 北京市朝阳区平乐园 100 号

独 创 性 声 明

本人声明所呈交的论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其它人已经发表或撰写过的研究成果，也不包含为获得北京工业大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

签名：__________ 日期：__________

关于论文使用授权的说明

本人完全了解北京工业大学有关保留、使用学位论文的规定，即：学校有权保留送交论文的复印件，允许论文被查阅和借阅；学校可以公布论文的全部或部分内容，可以采用影印、缩印或其它复制手段保存论文。

（保密的论文在解密后应遵守此规定）

签名：__________ 导师签名：__________ 日期：__________

摘 要

汽车内部电子控制器件的不断增加使网络化的汽车整体控制成为当前汽车电子产业的主要发展方向之一。CAN（Controller Area Network）总线由于其低成本、高可靠性、高抗干扰能力和高实时性等特点，成为汽车内部控制网络的主要载体。作为 CAN 总线的应用层协议之一，CANopen 具有高度的灵活性和可配置性，成为电动汽车和混合动力汽车控制网络的首选协议。

根据汽车系统的高实时性要求、多样的控制环境和 CANopen 协议本身的特点，建立 CANopen 网络的首要任务是建立一个实时运行、并行处理、灵活配置和可移植的 CANopen 主站。为了实现上述目标，本文提出了基于散列表的对象字典设计和基于标准 C 语言非抢占式任务调度机的 CANopen 主站协议栈设计。

基于散列表的对象字典有效地克服了传统数组型对象字典可配置性差的问题。同时通过对散列表的溢出表实施实时排序，大大加快了对象字典的读取速度。

基于标准 C 语言的非抢占式任务调度机为 CANopen 事件的并行处理提供了良好的平台，并天生具有良好的可移植性。在此之上建立的 CANopen 主站协议栈具有良好的实时性、高度的功能独立性和可移植性。

经过实际网络测试，该 CANopen 主站设计实现了最高 5kHz 的数据更新率，协议的完整度大大超过了开源协议栈。严格的代码结构也保证了较高的移植能力。

关键词　控制器局域网；CANopen 应用层协议；实时系统；调度算法；可移植

ABSTRACT

The trend to replace the mechanical parts in vehicle by microcontrollers makes the networklized systematic controlling an important research field in current vehicle research domain. As one of the main network platforms, CAN (Controller Area Network) bus is cheap, highly reliable, nosie-tolerant and real-time. Among the existed high level protocols based on CAN, the CANopen application layer protocol possesses excellent flexibility and configuration capability. Thus it is the first choice of nowadays control networks in electric vehicles and hybrid vehicles.

Since the internal network of vehicle is extremely a real-time and comprehensive control system, and CANopen protocol works in a extremely flexible way, the most important task to build a CANopen network is to design a real-time, parallel working, flexibaly configurable and transplantable CANopen master. To implement this master, an object dictionary architecture based on hash algorithm and a CANopen master protocol stack design based on a non-preemptive task scheduler are proposed.

In the architecture of object dictionary, the hash algorithm provides dynamic configuration capability, while traditional array based algorithms are impossible for run time configuration. By keeping overflow table in order, reading speed of the hash table is not obviously slower than that of array based methods.

Non-preemptive task scheduler provides a perfact platform to parallelly process multiple CANopen events, and it is inborn transplantable because the only requirement is the standard C library. Therefore, the CANopen master stack built on this possesses excellent real-timing, code independency and transplanting performance.

Proved by practical network, the CANopen master works at a maximal 5kHz data updating rate, provides more protocol compatibility. The rigidly partitioned source code also guarantees the easy transplanting to other platforms.

Keywords controller area network; CANopen application layer protocol; real-time system; scheduling algorithm; transplantable capability

目　　录

摘　要.. I
ABSTRACT.. II
第一章 绪 论.. 1
1.1 课题的研究背景 1
1.1.1 现场总线与汽车电子... 1
1.1.2 CAN 总线与 CANopen 应用层协议 2
1.2 课题的任务、难点及意义 4
1.3 论文结构 5
第二章 CAN 总线及 CANopen 应用层协议 7
2.1 简介 7
2.2 CAN 现场总线协议... 7
2.3 CANopen 应用层协议 10
2.4 CANopen 主站 13
2.5 本章小结 14
第三章 对象字典的设计和分析.. 15
3.1 简介 15
3.2 对象字典的特点 15
3.3 基于散列表的实现方法 16
3.3.1 传统的实现方法... 16
3.3.2 基于散列表的方法... 17
3.3.3 性能分析及速度优化... 25
3.4 本章小结 29
第四章 任务调度机的设计与分析.. 30
4.1 简介 30
4.2 任务调度方法的提出 30
4.2.1 任务调度的必要性... 30
4.2.2 调度算法的选择... 33
4.3 任务调度机的实现 34
4.3.1 任务的抽象... 34
4.3.2 调度算法... 36
4.4 性能分析 41
4.5 本章小结 43

第五章 基于调度机的 CANopen 主站协议栈设计 44
5.1 简介 44
5.2 协议栈的整体结构 44
5.3 驱动与报文队列 45
5.4 同步报文的生成 47
5.5 紧急报文处理 49
5.6 PDO 报文处理 50
5.6.1 PDO 报文的接收 50
5.6.2 同步 PDO 报文的发送 50
5.6.3 异步 PDO 报文的发送 51
5.7 SDO 报文处理 52
5.8 节点状态的维护 53
5.8.1 心跳报文机制 53
5.8.2 节点保护机制 53
5.9 网络启动过程 54
5.9.1 CANopen 主站的启动 55
5.9.2 通讯配置 55
5.9.3 从节点启动过程 56
5.10 本章小结 58
第六章 CANopen 主站的实现及测试 59
6.1 简介 59
6.2 主站测试平台 59
6.2.1 基于 WindowsXP 的 CANopen 主站设计 59
6.2.2 线程间的数据通讯 60
6.3 测试数据分析 63
6.3.1 实时性 63
6.3.2 WindowsXP 平台上 CANopen 主站的速度 64
6.3.3 WindowsXP 平台上 CANopen 主站的内存占用 65
6.3.4 节点启动检查 66
6.3.5 协议支持的完整性 68
6.4 关于可移植性的考虑 69
6.5 本章小结 71
结 论 72
参考文献 73
附 录 77

附录 1 初始对象字典......77
附录 2 新建从节点所需要的对象字典项......78
附录 3 对象字典散列算法搜索时间计算函数......79
附录 4 动态内存分配的简单实现......80
攻读硕士学位期间所发表的学术论文......83
致　　谢......85

第一章 绪 论

1.1 课题的研究背景

1.1.1 现场总线与汽车电子

上个世纪 70 年代，借助于数字计算机的发展，产生了基于集中控制的计算机控制系统。不过，集中控制系统需要收集所有设备的状态信息并分发控制信息，存在通讯延时较大、控制系统复杂和不可靠等缺点。因此，集中控制逐渐被分布式控制所取代。

与此同时，数字通讯网络的可靠性不断提高，使得在工业现场使用统一的标准化数字通讯网络成为可能。当然，使用于工业现场的控制网络必须具有高可靠性和高抗干扰能力。针对这些要求，很多国际组织和企业都提出了各自不同的总线标准，统称为现场总线，如德国西门子公司的 PROFIBUS，挪威的 Echelon 公司的 LonWorks，德国 BOSCH 公司的 CAN 等等。这些现场总线标准各有特点，在各自的应用领域得到了广泛的应用。

在汽车的控制系统中，电子装置正逐步取代机械部件，以微控制器为主体的汽车电子设备在整车中所占的比重不断升高。汽车正在由机械产品逐步转向机电一体化产品，进而转向以分布式网络技术为基础的智能化系统[1]。汽车电子化是现代汽车发展的重要标志之一。

汽车的电子化历程大概经历了三个阶段[2]。

第一阶段，用电子装置代替个别的机械功能部件，比如点火装置。

第二阶段，微型计算机开始在汽车上出现。电子控制设备和机械控制设备各有所长，并随着燃油资源和排气问题的出现，电子设备显示出其独特的优越性。机电一体化技术在该阶段迅速发展。

第三阶段，汽车电子向智能化、网络化的方向发展。大量的车内控制器由电子设备取代或者集成了电子控制单元，并且所有的电子控制器一起形成了一个协同工作的车内控制网络。在该网络下，网络控制、人工智能、模糊控制、计算机辅助驾驶等等控制技术与控制理念结合在一起。可以说，汽车的网络化是汽车电子走向成熟的标志之一。

针对于这种汽车的网络化控制趋势，多种使用于汽车控制系统的现场总线标准应运而生[3]，比如应用于低网络速率、成本较低的 Lin 总线，中网络速率的 CAN 总线和具有高网络速率的 FlexRay 总线等等。

1.1.2 CAN 总线与 CANopen 应用层协议

为了解决汽车内部众多控制器与测量设备之间的数据交换问题，德国 BOSCH 公司于 1986 年开发了一种新的串行数据通信总线——CAN（Controller Area Network）总线[4, 5]。

由于 CAN 总线使用了 11 位的标识符，并通过位同步机制，迫使低优先级的报文自动放弃对总线的驱动，实现了冲突避免机制。和基于冲突检测机制的现场总线相比，CAN 总线的冲突避免机制保证了在高网络负载情况下的高总线有效使用率，从而能够有效地支持分布式控制系统或实时控制系统，具有低成本、高可靠性、高抗干扰能力和高实时性等特点。此后，于 1993 年颁布的 ISO11898 国际标准[5]为 CAN 总线的规范化和应用系统设计铺平了道路。同时，各种基于 CAN 协议的高层协议开发使得 CAN 总线功能更强，应用范围更广，不仅在汽车工业、过程控制、数控机床和纺织机械等领域已取得广泛运用，而且正在向医疗、电力、海运电子设备等方面发展。

CAN 总线的设计初衷就是针对汽车控制网络的运用。在国际上，许多汽车公司早在 80 年代就积极致力于 CAN 总线技术的研究。从 1992 年起，Mercedes-Benz 公司已经开始在他们的高级客车上使用 CAN 技术。一个 CAN 总线对发动机进行管理，另外一个接收操作信号。这两个物理上独立的 CAN 总线系统，通过网关连接并实现数据交互。现在，继 Volvo、Saab、Volkswagen 和 BMW 之后，Renault 和 Fiat 也开始在他们的汽车上使用 CAN 总线。在欧洲，几乎每一辆新客车上均装配有 CAN 总线[6]。

然而，CAN 本身是一个底层协议，仅详细定义了物理层和数据链路层，本身并不完整。很多复杂的应用问题需要更高层次的定义来解决。比如，CAN 数据帧一次最多只能传送 8 字节，而不能传输大于 8 字节的长帧；CAN 只提供了非确认的数据传输服务，而无法提供有确认的数据传输，等等。所以，CAN 协议允许各厂商在 CAN 物理层的基础上自行开发高层应用协议，以满足不同应用的需要。这种做法使得 CAN 能够灵活地运用到工业控制的各个领域，但是也造成了 CAN 应用层协议的不统一。

在这种情况下，1992 年在德国成立了名为自动化 CAN 用户和制造商协会（CiA, CAN In Automation）的非赢利组织。该协会制定并标准化了一系列的 CAN 高层协议，并向其用户提供所有关于 CAN 的标准、使用和测试的咨询服务。1993

年，在 CAN 的开发者——德国 BOSCH 公司的带领下，一个基于 CAN 基本协议的应用层协议规范——CANopen 协议在欧盟的资助项目中设计并提出[7]。

CANopen 协议着重定义了应用层以及相关的通讯架构，详细内容包括对象字典、网络管理、启动配置、各种传输对象的定义等等。

其中，对象字典是 CANopen 的关键，它保存了一个 CANopen 节点所有的配置参数和通讯数据，也提供了 CANopen 应用层和用户程序交流的接口。正是由于对象字典的存在，在 CAN 总线上传输的报文不需要包含所传数据的格式定义、类型与作用等附加信息，只需包含实际的数据。报文的接收端只需借助对象字典的帮助，便可以解析 CAN 报文内的信息，因为 CAN 报文中的每一个比特都被对象字典完全定义。所以，CANopen 协议具有很高的数据传输效率。

网络管理和启动配置则体现了 CANopen 协议的灵活性。CANopen 协议并不要求在启动网络时预知网络中的所有节点信息。根据启动配置过程，CANopen 网络可以主动发现网络中的节点，并正确地检查和配置新节点。网络管理则提供了灵活的节点维护机制。整个 CANopen 当中的所有节点都可以被其它节点监视和控制，实现网络的动态管理。

各种传输对象则扩展了 CAN 总线标准中对于标识符的定义。不同类型的通讯对象完成不同的通讯任务，通讯任务的优先级则由标识符的优先级直接标识。对象字典中对于不同通讯对象的配置则保证了该机制的高通讯效率。

总的来说，CANopen 协议是一个具有高度灵活性和高通讯效率的 CAN 应用层协议。

当然，CANopen 并不是唯一被 CiA 认可的 CAN 应用层协议。同样运用于 CAN 网络的应用层协议还有 SAE J1939、DeviceNet、CANKingdom 和 SDS。它们各有特点。SAE J1939 协议主要用于重型卡车控制系统，并使用了 CAN 2.0 扩展协议，针对于重型卡车系统的通讯要求对标识符进行了重新定义，但是数据报文中包含了数据的类型和格式信息，便于解析但是通讯效率不高。DeviceNet 和 CANopen 类似，也具有相似的对象字典结构。不过 DeviceNet 的定义更为复杂和严格，有严格层次定义，节点开发成本较高。CANKingdom 采用通用模块化的设计方法使得各种模块都能很容易地接入 CANKingdom 网络当中。SDS 适用于高效实时的系统，采用标准的主从结构，由主控节点进行全权控制。

相比之下，CANopen 是一个完全开放的协议，开发者可以免费获得协议授权，支持 CAN1.0 基本和 CAN2.0 扩展协议，支持从 20kbps 到 1Mbps 的多种传输速率，拥有和 OSI 兼容的基本架构，具有高度的灵活性。相比 DeviceNet 协议，CANopen 需要的代码量和运算量较小，特别适合于中小型的嵌入式系统[8]。

1.2 课题的任务、难点及意义

本课题父项目的最终目标是建立混合动力汽车的 CANopen 控制网络。在这个网络中，混合动力汽车的每一个功能部件为一个 CANopen 节点。由于整个混合动力汽车的功能部件正在研发当中，所以本课题的实际任务是建立一个具有高度实时性、灵活性和可移植性的 CANopen 主站协议栈。在混合动力汽车的各个功能部件完成之后，该平台应当能够被容易地移植到实际汽车整车控制系统中，控制和监视整个混合动力汽车内部控制系统。

该课题的难点主要体现在两个方面，提高主站的实时性和高度可配置的对象字典实现。运用于汽车控制系统的 CANopen 网络必须具有高度的实时性。尽管 CANopen 相比其它的 CAN 应用层协议具有通讯效率较高和实现简单的特点，然而如何能够在资源紧张的嵌入式系统中提高系统实时性仍然是一个关键问题。另外，CANopen 完全依赖于对象字典来控制通讯并提供应用软件接口。CANopen 主站的管理与配置能力需要动态可配置的对象字典支持。传统的基于数组的对象字典实现方式难以提供高度的可配置性，因而需要新的实现方式来改造现有对象字典。

从现阶段的研究成果来看，国内有大量的 CAN 基本应用研究。和传统的基于 485 通讯协议的串行总线相比，CAN 总线具有更高的数据传输率和抗干扰能力，所以大量的 485 控制系统被 CAN 总线控制系统所取代。国内大部分的 CAN 研究还停留在如何更好地使用 CAN 总线来设计现场总线控制系统。

相比之下，关于 CAN 总线的应用层协议的研究就要少一些。关于 CANopen 的研究都停留在如何将 CANopen 节点使用到现有的控制网络当中。这些研究一般基于国外的商业节点和商业控制与仿真软件，强调这些节点的使用，而非如何设计一个 CANopen 节点。

从 1999 年起，中国单片机公共实验室（BOL）开始了对 CANopen 和 SAE J1939 标准的研究工作，但是除了中国单片机公共实验室之外，鲜有其它的研究机构进行相关深入的研究。在大学研究院所中，仅有天津大学和北京工业大学在此方面有相应突出的研究成果。但是这些研究也停留在改造国外的开源 CANopen 协议栈实现 CANopen 从节点的水平上。

比如，天津大学宋晓强同学的硕士学位论文“CAN bus 高层协议 CANopen 的研究以及在模块化 CAN 控制器上的实现[9]”在单片机上实现了一个 CANopen 从节点。但是其二级总线的结构限制了节点的实时性。

天津大学王宇波同学的硕士学位论文“嵌入式网络控制器的设计与实现[10]”将美国嵌入式系统协会提供的 Micro CANopen 协议栈[11]实现在了 AT91RM9200 嵌入式处理器平台上。但是 Micro CANopen 协议栈本身并不是一个完整的

CANopen 从节点协议栈，仅支持部分的对象字典功能和最多 8 个静态配置的过程通讯对象。其从节点的通讯功能和可配置性受到很大影响。

北京工业大学陈涛同学的工学硕士学位论文“汽车仪表的 CANopen 节点通信的研究与实现[12]”将受开源社区项目支持的 CanFestival CANopen 协议栈移植到了自主设计的 ARM9 平台。但是经过实际测试发现，CanFestival CANopen 协议栈的实时性较差，通讯速度缓慢。其基于数组的对象字典实现形式也制约了节点的可配置能力。

北京工业大学肖进军同学的工学硕士学位论文“混合动力电动汽车 CANopen 总线协议的研究与实现[13]”改造了 Microchip 公司提供的 CANopen 从节点协议栈，并将其移植到了 TMS320F2812 平台，实现了一个功能较为完整的实时 CANopen 电动机节点。不过，Microchip 公司的 CANopen 从节点协议栈通过代码的方式静态支持过程数据通讯，可配置性较弱。

天津大学陈骥同学的硕士学位论文“基于 CANopen 高级协议和 ED 调度算法的电动汽车网络协议研究[14]”利用 PCI-CAN 转接卡在 PC 机上实现了一个不具备对象字典和动态可配置性的 CANopen 监控主站。从论文的测试结果来看，5ms 的通讯周期显然较大，实时性不高，难以使用于汽车控制系统。

在国际上，在 CANopen 协议提出之后，CANopen 协议栈的开发主要由现场总线开发商推动。由于知识产权和产品化的问题，鲜有关于实现 CANopen 协议栈的学术文章发表。大部分的 CANopen 相关文章也都集中在如何在各种控制系统中使用商业化的 CANopen 节点。

以 CanFestival 为首的开源 CANopen 项目在近年逐步发展。不过在当前阶段，开源的 CANopen 协议栈大多功能不全、实时性较差或不具备可配置性。所以，本课题所设计的具备高度实时性、灵活性和可移植性的 CANopen 主站平台在国内还是一个空白，具有极高的实用价值和现实意义。

1.3 论文结构

本文将以如下的方式展开：

第二章将首先介绍 CAN 现场总线协议，CANopen 应用层协议规范。在此基础上，讨论 CANopen 主站的具体定义、运行特点和实现要求。

第三章将分析 CANopen 对象字典的定义和逻辑结构。针对于 CANopen 主站对对象字典的独特要求，提出一种基于散列表的对象字典实现方案。

第四章将分析 CANopen 主站的运行要求，提出一种基于标准 C 语言，具有高度实时性和可移植能力的任务调度机机制。

第五章将给出 CANopen 主站协议栈的实现细节。按照 CANopen 主站的各种功能要求，逐一分析各功能基于调度机机制的实现方法。

第六章将针对在 WindowsXP 平台上建立的 CANopen 主站测试网络，给出本文所述 CANopen 主站的运行性能。

第二章 CAN 总线及 CANopen 应用层协议

2.1 简介

本章首先将介绍 CAN 总线协议，CAN 总线的基本性能，以及 CAN 节点的硬件结构；然后分析 CANopen 应用层协议、CANopen 节点的状态机和基本结构；最后将给出 CANopen 主站的概念。

2.2 CAN 现场总线协议

CAN 总线[5, 15]是由德国 BOSCH 公司在 1986 首先提出的一种用于汽车控制，有效支持分布式控制和实时控制的串行通讯总线协议。

该协议定义了 CAN 总线的物理层和数据链路层，其它部分的实现由用户定义，因此是一个未完全定义的协议。图 2-1 定义了 CAN 标准报文的基本结构[16]，该结构也决定了 CAN 总线的通讯特点。

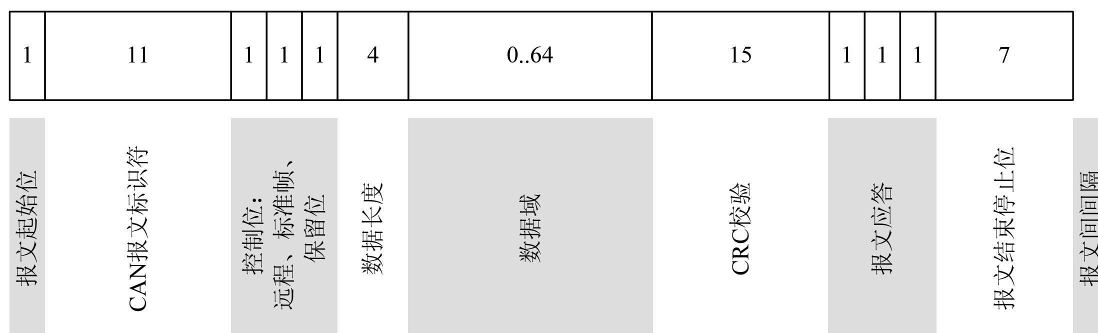


图 2-1 CAN 标准报文

Fig. 2-1 the CAN Base Frame Format

CAN 总线协议具有严格的报文优先级。该优先级由报文的 11 位标识符决定。同时 CAN 总线协议采用了载波侦听多路访问冲突避免（CSMA/CA）方式[16, 17]。CAN 通讯在总线上的所有节点之间建立位同步。这样，当两个报文同时

发出，发生竞争总线现象时，由于位同步的存在，低优先级报文的发送方可在位时间内获知总线竞争失败，从而让高优先级的报文获得无冲突的传输。

位同步保证了高优先级报文的优先传输，但也限制了 CAN 网络节点间的最大传输速率。由于位同步，单比特在 CAN 总线上的存活时间就必须大于总线上相距最远的两个节点之间的传输延时。从而，CAN 总线的总线长度决定了数据通讯的最高速率，如图 2-2[7,16]。

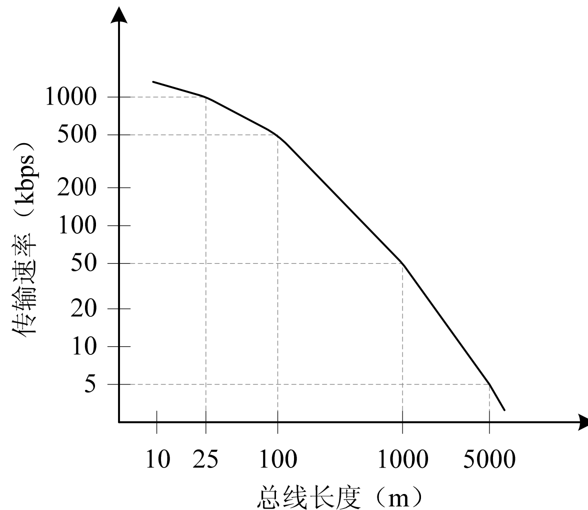


图 2-2 CAN 网络总线长度与传输速率关系

Fig. 2-2 CAN Bus Bit Rate versus Bus Length

混合动力汽车控制网络的总线长度在 25m 以下，最高总线速率可达 1Mbps。考虑到一个报文的有效载荷最多为 64 比特，占报文长度的 59.26%。在最大传输速率 1Mbps 时，实际数据传输速率小于 600kbps，远远低于现已大量运用的 100M 以太网[18]。和基于令牌的工业现场总线协议 PROFIBUS 和 ControlNet 比较，CAN 总线也不具备在高数据负载情况下高效数据传输。所以，CAN 总线是一种基于短数据报文优化的，适用于低数据负载的实时控制网络[19]。可见，CAN 总线适合于实时控制网络，但并不适合于数据传输。

此外，尽管采用了 CSMA/CA 机制，CAN 总线仍然会受到各种干扰的影响，产生通讯错误。为了缓解该问题，CAN 总线的数据链路层提供了一套完整的报文错误检测和重传机制。

CAN 节点的数据链路层会接收 CAN 总线上的每一个报文，自动计算报文的校验码，并和报文中的 CRC 校验比对，返回正确的应答或者返回错误的应答以迫使报文重传。为了防止在网络拥塞或者网络连接失效的情况下进行无效的重传尝试，CAN 节点会按图 2-3 所示的状态机[15, 16]控制节点的运行状态。

上电后，CAN 节点的接收错误计数器和发送错误计数器都为 0，处于错误检测状态。每一次正确接收或发送一个 CAN 报文，相应计数器减一；如果错误，则相应增一。在错误检测状态，如果 CRC 校验失败，CAN 节点会主动发送错误应答，迫使报文重传。如果总线错误过多，接收或发送错误计数器大于 127，节点转入被动接收状态。在该状态中，节点虽然能够正常接收和发送报文，但不会根据 CRC 校验而发送错误应答。如若错误次数继续增加，发送错误计数器大于 255，节点会自动关闭，所有接收和发送功能终止，需等待上层重新启动[16]。由于错误计数器在成功地接收和发送之后会逐步递减，正常情况下，节点应当工作于错误检测状态。

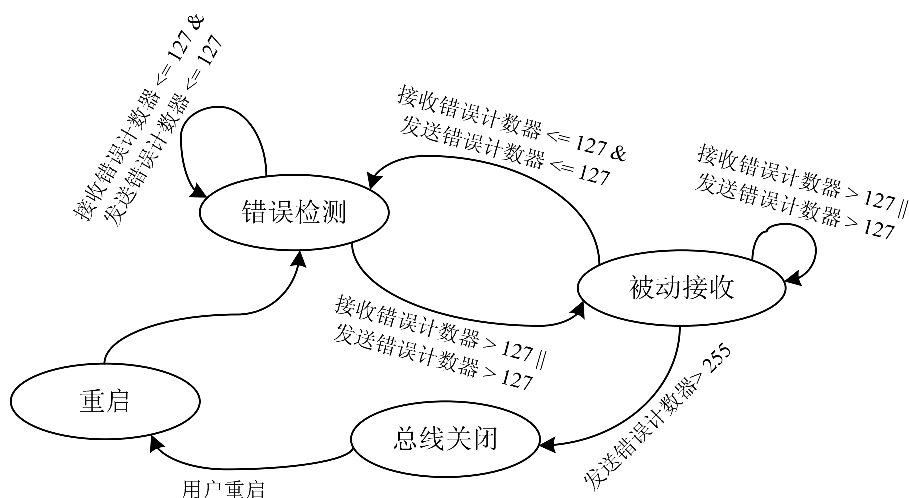


图 2-3 CAN 节点数据链路层状态转换图

Fig. 2-3 States of the Data Link Layer in a CAN Node

基于 CAN 总线物理层和数据链路层协议，CAN 节点的硬件结构一般如图 2-4 所示。

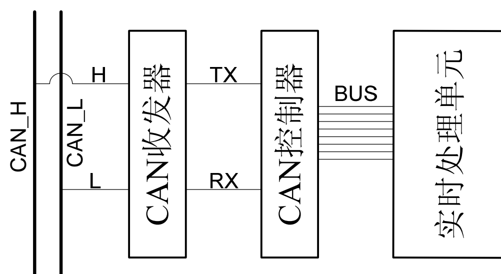


图 2-4 CAN 节点的硬件结构

Fig. 2-4 the Achitecture of a CAN Node

CAN 收发器和 CAN 控制器通常为单独的芯片。其中 CAN 收发器负责 CAN 总线电平和节点内部电平之间的转换，完成 CAN 物理层的功能。CAN 控制器则负责发送和接收报文，解析报文，并且根据 CRC 校验的结果维持数据链路层状态机。CAN 节点数据链路层以上的功能由承载用户定义应用层的实时处理单元完成。

2.3 CANopen 应用层协议

如前所述，CAN 总线协议仅规定了物理层和数据链路层，而将其它的高层留给用户定义，这样便造成了兼容问题。两个 CAN 节点只有在使用相同的高层协议时才能正常通讯。

为了解决该问题，很多组织提出了不同的 CAN 应用层协议规范，比如用于重型卡车的 SAE J1939 协议[20]，用于仪器的 DeviceNet[21]和开源的 CANopen[7]。相比较而言，CANopen 协议不针对某种特别的应用对象，具有较高的配置灵活性，高数据传输能力，较低的实现复杂度。同时，CANopen 完全基于 CAN 标准报文格式，而无需扩展报文的支持，最多支持 127 个节点，并且协议开源。因此本课题选用 CANopen 作为混合动力汽车内部的 CAN 控制网络应用层协议。

一个标准的 CANopen 节点（图 2-5），在数据链路层之上，添加了应用层。该应用层一般由软件实现，和控制算法共同运行在实时处理单元内。

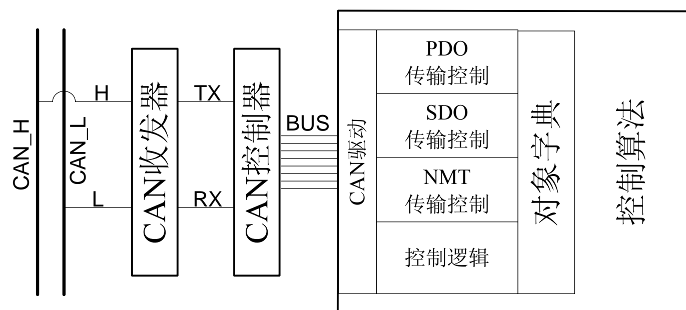


图 2-5 CANopen 节点的结构

Fig. 2-5 Architecture of a CANopen Node

CANopen 应用层协议细化了 CAN 总线协议中关于标识符的定义[7]。定义标准报文的 11 比特标识符中高 4 比特为功能码，后 7 比特为节点号，重命名为通讯对象标识符（COB-ID）。功能码将所有的报文分为 7 个优先级，按照优先级从高至低依次为：网络命令报文（NMT）、同步报文（SYNC）、紧急报文

（EMERGENCY）、时间戳（TIME）、过程数据对象（PDO）、服务数据对象（SDO）和节点状态报文（NMT Err Control）。7 位的节点号则表明 CANopen 网络最多可支持 127 个节点共存（0 号节点为主站）。表 2-1 给出了各报文的 COB-ID 范围。

表 2-1 CANopen 的 COB-ID 分配

Table 2-1 COB-ID Allocation of CANopen

报文类型	功能码	**COB-ID 范围(Hex)**
NMT	0000	000h
SYNC	0001	080h
EMERGENCY	0001	081h~0FFh
TIME	0010	100h
PDO1(发送)	0011	181h~1FFh
PDO1(接收)	0100	201h~27Fh
PDO2(发送)	0101	281h~2FFh
PDO2(接收)	0110	301h~37Fh
PDO3(发送)	0111	381h~3FFh
PDO3(接收)	1000	401h~47Fh
PDO4(发送)	1001	481h~4FFh
PDO4(接收)	1010	501h~57Fh
SDO(发送)	1011	581h~5FFh
SDO(接收)	1100	601h~67Fh
NMT Error Control	1110	701h~77Fh

NMT 命令为最高优先级报文，由 CANopen 主站发出，用以更改从节点的运行状态。SYNC 报文定期由 CANopen 主站发出，所有的同步 PDO 根据 SYNC 报文发送。EMERGENCY 报文由出现紧急状态的从节点发出，任何具备紧急事件监控与处理能力的节点会接收并处理紧急报文。TIME 报文由 CANopen 主站发出，用于同步所有从站的内部时钟。PDO 分为 4 对发送和接收 PDO，每一个节点默认拥有 4 对发送 PDO 和接收 PDO，用于过程数据的传递。SDO 分为发送 SDO 和接收 SDO，用于读写对象字典。优先级最低的为 NMT Error Control 报文，由从节点发出，用以监测从节点的运行状态。

每个节点维护一个对象字典（Object Dictionary, OD）。该对象字典保存了节点信息、通讯参数和所有的过程数据，是 CANopen 节点的核心数据结构。同时，上层应用程序也主要通过读写对象字典和 CANopen 应用层进行交互。

CANopen 对象字典为两级数组结构。第一级数组称为主索引，宽度为 FFFFh。每一个主索引可拥有一个宽度为 FFh 的子索引表。因为 CANopen 对象

字典支持的索引范围巨大（如果将子索引和主索引一同考虑，CANopen 对象字典支持约 16.8M 个索引），CANopen 对象字典的实现也是 CANopen 应用层开发的一个难点。不过，并非所有索引都需实现，一个节点只需实现能完成功能的最小对象字典集合就可正常工作。根据 CANopen 协议，表 2-2 定义了主索引的数据分布。

表 2-2 主索引数据分布

Table 2-2 Mapping of Main Index

索引	对象
0000h	保留
0001h~001Fh	基本的数据类型
0020h~003Fh	复杂的数据类型
0040h~005Fh	生产商相关数据类型
0060h~007Fh	设备描述的基本的数据类型
0080h~009Fh	设备描述的复杂数据类型
00A0h~0FFFh	保留
1000h~1FFFh	通讯参数
2000h~5FFFh	制造商的特殊设备描述文件
6000h~9FFFh	标准设备描述文件
A000h~BFFFh	标准接口描述文件
C000h~FFFFh	保留

根据节点所支持的通讯方式，每一个节点都必须实现 1000h~1FFFh 当中的一个必要子集，同时实现 6000h 以上部分的数据区。数据区大小由节点的功能自行决定。1000h 以下部分所有的节点都不需实现。

另外，CANopen 的每一个节点都维护了一个状态机（图 2-6[7]）。该状态机的状态决定了该节点当前支持的通讯方式以及节点行为。初始化时，节点将自动设置自身参数和 CANopen 对象字典，发出节点启动报文，并不接收任何网络报文。初始化完成后，自动进入预运行状态。在该状态，节点等待主站的网络命令，接收主站的配置请求，因此可以接收和发送除了 PDO 以外的所有报文。运行状态为节点的正常工作状态，接收并发送所有通讯报文。停止状态为一种临时状态，只能接收主站的网络命令，以恢复运行或者重新启动。表 2-3 列出了各状态之间转换的具体条件。

2.4 CANopen 主站

在 CANopen 应用层协议规范[7]的基础上，CANopen 管理者框架协议[22]进一步规定了网络管理主站（NMT Master）、配置管理者（Configuration Manager）和服务数据对象管理者（SDO Manager）的功能和行为，以及详细的从节点启动过程，完善了应用层协议。

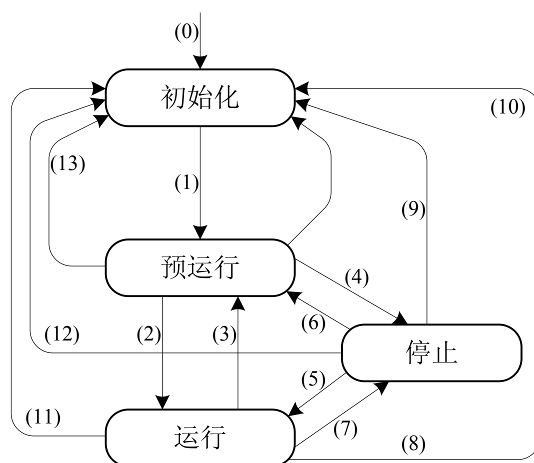


图 2-6 CANopen 节点状态转换图

Fig. 2-6 State Diagram of a CANopen Node

表 2-3 CANopen 节点状态转换条件

Table 2-3 Conditions of State Change

(0)	设备上电
(1)	设备初始化完成
(2)(5)	收到启动节点命令
(3)(6)	收到进入预运行命令
(4)(7)	收到停止节点命令
(8)(9)(10)	收到重启节点命令
(11)(12)(13)	收到重启通讯命令

作为网络管理主站，CANopen 负责监控并检查 CANopen 网络当中所有节点的运行状态。其中包括在启动配置时，根据 CANopen 管理者框架协议启动所有的必须节点，尝试寻找和启动可选节点。在运行过程中，监控网络中节点的运行状态，及时处理节点的异常状态变化。作为配置管理者，CANopen 主站检查各个从站的设备配置文件（DCF），管理、保存和下载各节点的电子数据表

（EDS）。作为服务数据对象管理者，CANopen 主站动态管理并建立从节点之间的 SDO 通道。

针对于混合动力汽车的 CANopen 主站，混合动力汽车的控制系统都基于嵌入式控制器，属于内存紧缺型设备，因而并无存储和动态更新 DCF 与 EDS 文件的要求。所以，本课题的 CANopen 主站并不实现配置管理者的功能。

2.5 本章小结

CAN 总线协议所具有的严格优先级特性以及它的冲突避免机制使得 CAN 总线非常适合作为实时设备的控制总线，然而高层协议的缺乏导致各种应用的不兼容。CANopen 作为一种 CAN 总线的高层协议提供了较为灵活的管理方式和高效的通讯方式。应用于混合动力汽车控制系统的 CANopen 网络需要 CANopen 主站来实时管理和监控网络。针对于混合动力汽车的实际应用需要，本文的 CANopen 主站设计任务是实现一个具备所有 CANopen 节点功能的网络管理主站。

第三章 对象字典的设计和分析

3.1 简介

本章将从 CANopen 主站和从站对象字典的差异出发，分析传统的基于数组的对象字典实现方式的局限性。为了解决这种局限性，本章将提出一种基于散列表的对象字典实现方法，并根据 CANopen 主站的数据特征确定实现参数。最后，本章将根据仿真结果分析该方法的性能，并最终提出一种改进的散列表方法。

3.2 对象字典的特点

表 2-2 已经给出了 CANopen 对象字典的数据分布。对于一个 CANopen 从站来说，对象字典由两部分构成：CANopen 应用层协议规范[7]规定的设备参数区（1000h~100Ah）、基本参数（100Ch~1017h）、SDO 与 PDO 配置（1200h~1BFFh)；具体从节点的数据区定义，例如 I/O 类型节点的 DS401 协议[23]。所有这些部分在节点的设计阶段可以完全确定，因此在运行时不需要动态更新对象字典结构。

CANopen 主站的对象字典除了和从节点相同的部分外，还需实现 CANopen 管理者框架协议[22]规定的 1F00h~1F91h 对象字典项。和从节点对象字典相比，CANopen 主站对象字典最大的特点在于其动态可配置的特性。由于 CANopen 主站需要管理整个 CANopen 网络，而在设计阶段，网络当中的节点个数、节点类型、数据区需求等等数据都无法确定，而需在运行时动态配置。根据 CANopen 对象字典的定义，预先保留这些数据空间将产生大量的数据区浪费。嵌入式控制器的硬件特点和这种大数据区的需求相矛盾，因而必须提供一种机制能够在保证对象字典项搜索速度的同时，提供字典项的动态添加和删除特性。这也是 CANopen 主站的特殊要求[24]。

3.3 基于散列表的实现方法

3.3.1 传统的实现方法

现有的开源 CANopen 从站和主站[11, 25~27]对象字典的实现都是基于数组和数组的变型。

由于 CANopen 对象字典的主索引和子索引结构形成天然的二维数组。通过主索引和子索引得到的数组下标在内存中就表示为索引数据距离对象字典首地址的偏移量。通过该偏移量，可直接访问对象字典项数据。因而访问一个对象字典项的算法为访问时间固定的 $O(1)$算法。另外，考虑到 CANopen 从节点的对象字典固定，不需要动态添加和删除索引，数组确实是 CANopen 从站对象字典的最好实现方式。

不过，CANopen 从节点的对象字典主索引并不连续，也并不是每一个主索引都有 256 个子索引，完全按照二维数组的方式实现对象字典必然造成大量的数据空间浪费。所以，所有的开源 CANopen 从站都采用了数组变型的实现方式。

以开源社区支持的 CanFestival 开源 CANopen 主站[25]为例，CanFestival 使用如表 3-1 所示的映射表实现了对不连续对象字典的连续存储。

表 3-1 主索引内存映射表

Table 3-1 Memory mapping table of main index

主索引	内存存储索引
1000h	0
1001h	1
1005h	2
1006h	3
1010h	4
1011h	5

首先，不连续的主索引按照连续的方式定义在数据内存中，各主索引的实际数组下标由内存存储索引表示。然后，对象字典的搜索函数在代码空间保存该内存存储索引和实际主索引的映射（表 3-1）。当需要访问一个主索引时，搜索函数通过查询表 3-1，得到实际内存存储索引，进而访问该主索引。但是由于映射表必须采用线性搜索的方式，实际上已经从 $O(1)$算法退化为 $O(n)$算法。

即便如此，由于映射表保存在代码空间，并且主索引在内存中的存储数组在运行时无法改变空间大小，该方法不能动态添加和删除索引。显然与主站对象字典需要支持索引项的动态添加和删除的要求矛盾。

另外一个由开源社区项目 OCERA 实现的 CANopen 主站[27]，也采用数组变型的方式。然而，该项目的所有子工程建立在嵌入式 RTLinux 操作系统上。CANopen 主站自然也建立在 RTLinux 的环境当中。该主站可以提供一定程度的针对于主索引的动态添加和删除，但完全依赖于操作系统的动态内存分配方法：每一个新索引项的建立都会调用内存分配函数。因此，该方法虽能在运行时动态配置对象字典，但是添加和删除单个索引的操作复杂，时间较长，容易产生内存碎片，并完全依赖于操作系统。

此外，基于特定操作系统的商业 CANopen 主站一般使用数据库实现对象字典。当然，在有操作系统支持的情况下，数据库能够近乎完美地实现对象字典的所有功能。但是数据库本身要求大量的内存空间、操作系统和高速处理器的支持，并很难达到实时控制，和混合动力汽车的控制系统要求相违背。

综上所述，数组变型的对象字典实现方法具有实现简单，访问时间快的优点，但是不能动态添加和删除索引，不符合 CANopen 主站的基本要求。直接利用操作系统的动态内存分配能够实现动态添加和删除索引，但是容易造成内存碎片，并且速度较慢。利用数据库实现对象字典能够解决动态配置的相关问题，但是增加了系统负担，不符合混合动力汽车控制系统的系统要求。

3.3.2 基于散列表的方法

由于数组变型的实现方法不能满足 CANopen 主站对象字典的运行时配置要求，本文提出一种基于散列表实现对象字典的方法。

3.3.2.1 主索引与子索引的归一化

根据 CANopen 应用层协议规范[7]关于对象字典索引项的定义，对象字典索引项以图 3-1 所示的格式表示。

索引号	节点类型	节点名称	数据类型	读写类型	必须/可选

图 3-1 对象字典项格式

Fig. 3-1 Format of OD Entry

显然，从实现的角度来说，并不是所有的信息都必须用代码体现，而只需要实现有用的信息。图 3-1 当中节点类型和数据类型信息重复，对于 CANopen

节点的实现来说，数据类型已经足够。同时“必须/可选”信息也不用在代码中体现。另外，这里的索引号并没有规定是主索引还是子索引。实际上，当一个主索引包含子索引的时候，该主索引的数据类型为数组（Array），并且该主索引的 0 号子索引默认为数组长度，而不包含任何数据。这便提供了一种将主索引和子索引归一化的可能。

当一个主索引为数组类型的时候，在数据类型当中标注该索引为数组类型，而在数据区记录数组长度，则能将数组类型的主索引和 0 号子索引合并。进一步，扩展索引号：在每一个索引项当中记录主索引和子索引。如果该索引为主索引，子索引域为 0。这样，从数据结构的角度，便不存在主索引和子索引的区别，将 CANopen 对象字典的二维数组结构转化为一维数组。

应当注意到，图 3-1 仅是一个描述格式，并没有包含数据区，然而代码实现时却需要。根据以上的归一化方法，确定索引项的数据结构如图 3-2。

```
typedef struct _ODIndex {
    short index;                    // 主索引
    char  subIndex;                 // 子索引
    char  indexType;                // 索引类型
    int data;                       // 数据
    // 执行函数
    char (*pFun)(struct _ODIndex *, char, int);
    struct _ODIndex * next;         // 链表指针
} ODIndex, *pODIndex;
```

图 3-2 索引项的数据结构

Fig. 3-2 Data Structure of OD Entry

在图 3-2 中，*index* 和 *subIndex* 分别保存了主索引和子索引。*indexType* 保存了该索引项的数据类型和访问类型信息，具体定义由表 3-2 给出。真正的数据保存在 *data* 当中，这里用一个 32 比特整型代表数据，实际类型由 *indexType* 决定。*pFun* 为一个函数指针；某些数据字典项具有特定的功能，这些功能由 *pFun* 指向的函数完成。链表指针 *next* 构成了溢出表，和散列表方法的溢出表相关，将在后文叙述。

3.3.2.2 散列表算法

通过归一化 CANopen 索引项，CANopen 对象字典由二维数组降维为一维数组。数组的地址空间为 24 比特（主索引 16 比特和子索引 8 比特），包含有效索引 16M，远大于实际需要的索引个数。

根据散列表原理，我们可以将主索引和子索引当作输入 *addr*，送入一个散列公式 $h_addr = hash(addr)$，得到散列之后的地址 *h_addr*。如果 *h_addr* 的地址范围比 *addr* 的地址范围小，那么就是将一个大空间映射到一个小空间[28]。正是由于该特性，散列表可以将大范围不连续分布的对象字典保存在紧致连续分布的内存范围内。

表 3-2 索引项类型定义

Table 3-2 Definitions of the Type of OD Entry

比特位置	子类型	意义
[4..0] 数据类型	0x00	索引未使用
	0x01	字节
	0x02	短整
	0x03	整型
	0x04	有符号整型
	0x05	字符串
	0x06	数组
	其它	未定义
[5] PDO 映射	0x1	索引映射至异步 PDO
	0x0	其它
[7..6] 读写类型	0x0	未定义
	0x1	只读
	0x2	只写
	0x3	读写

当然，在这种过程中可能出现冲突，即多个 *addr* 在经过散列计算之后得到相同的 *h_addr*。解决冲突的办法有多种[28, 29]：

顺序填充法。如图 3-3，记录 10 的散列后地址 *h_addr* 与记录 2 冲突，则从 *h_addr* 开始向后寻找，直至找到未使用的地址 4，填充该记录。

再散列法。如图 3-4，按照散列公式 $hash(addr) = addr\%8$，记录 10 的散列后地址为 2，与记录 2 冲突。将散列结果放入散列公式 $hash'$ 重新计算，得到新的地址 6，冲突解决。如果重新散列后的结果仍然产生冲突，则将上一回的计算结果放入散列公式 $hash'$ 再次迭代，直至得到一个无冲突地址。

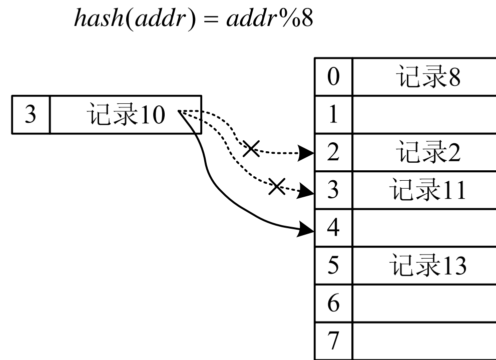


图 3-3 顺序填充法

Fig. 3-3 Sequential Filling Method

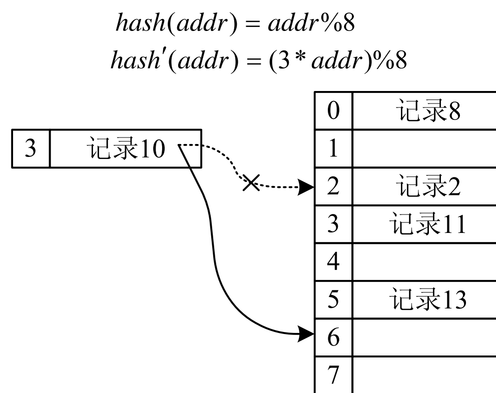


图 3-4 再散列法

Fig. 3-4 Double Hash Method

公共溢出区法。该方法使用一个公共的记录表来保存所有出现冲突的记录。如图 3-5，记录 10 的散列后地址与记录 2 产生冲突，则在公共溢出区依次寻找到一个未使用的地址，填充该纪录。

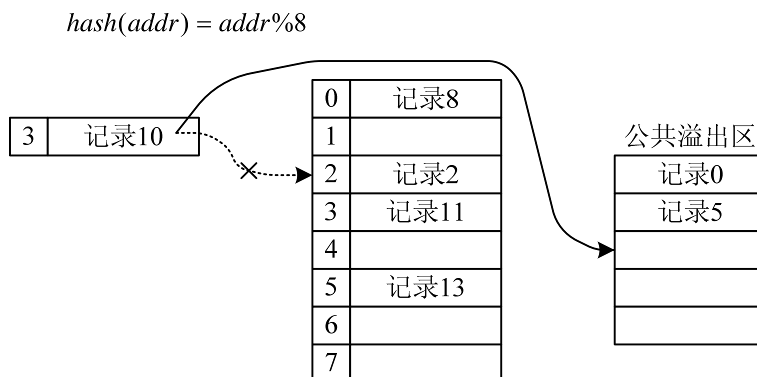


图 3-5 公共溢出区法

Fig. 3-5 Public Overflow Table Method

链表法。该方法把所有映射到同一个 *h_addr* 地址的项组成一个链表，称为溢出表。如图 3-6，记录 10 和记录 2 发生冲突，则该记录被保存在记录 2 指向的溢出表内。

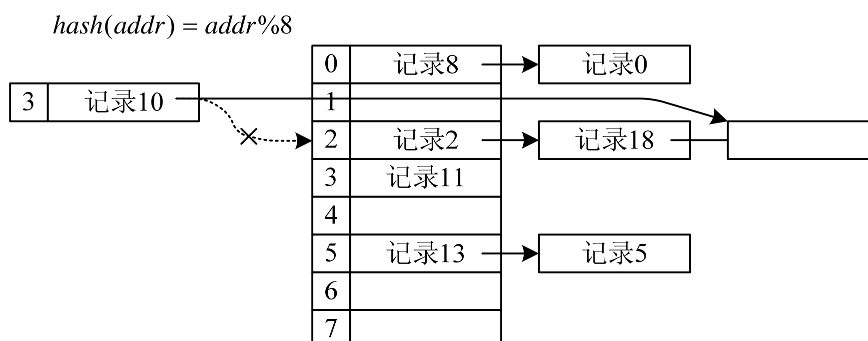


图 3-6 链表法

Fig. 3-6 Linked List Method

顺序填充和再散列法都假设出现冲突后，经过多次寻找总能找到一个不出现冲突的 *h_addr*。但是，无论 *h_addr* 的地址范围为多大，只要小于 CANopen 的最大可能字典项个数，就不能保证能容纳所有的对象字典项。因此顺序填充和再散列方法并不适用。

公共溢出区法不用保证散列表的地址空间必须能保存所有的数据，然而，它假设发生冲突的项不是很多。如果冲突不断发生，会导致公共溢出区不断增大。由于公共溢出区为无序保存，搜索时间和数据项的个数成正比关系，搜索时间将随着冲突的发生而线性增长。

链表法对散列表容量和数据项个数没有任何要求。唯一的问题是当冲突大量出现，溢出表长度也和冲突个数成正比，因此搜索时间也将线性增长。但是，和公共溢出区法相比，散列表法将冲突项分散到各个地址上的溢出表中，溢出表的平均长度将远小于公共溢出区法。

鉴于 CANopen 协议的高度灵活性，主站不应对对象字典的最大容量做任何假设，因此，我们选择链表法作为散列表的冲突处理方法。图 3-7、3-8 和 3-9 分别给出了在散列表中建立一个索引项、搜索一个索引项和删除一个索引项的伪算法。

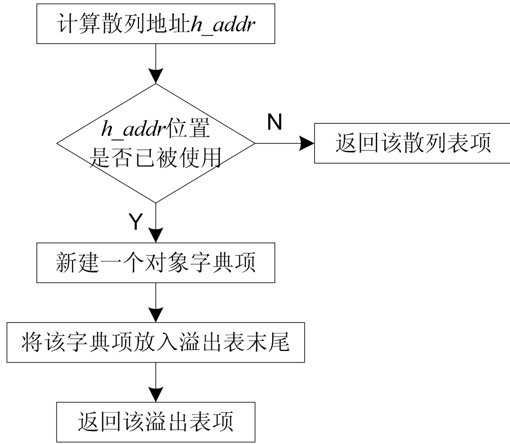


图 3-7 添加一个对象字典项

Fig. 3-7 Algorithm of Adding a New OD Entry

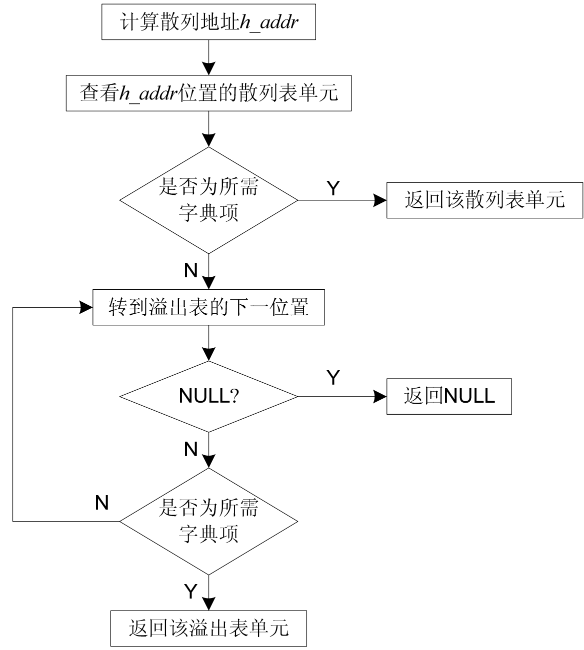


图 3-8 搜索一个索引项

Fig. 3-8 Algorithm of Searching an OD Entry

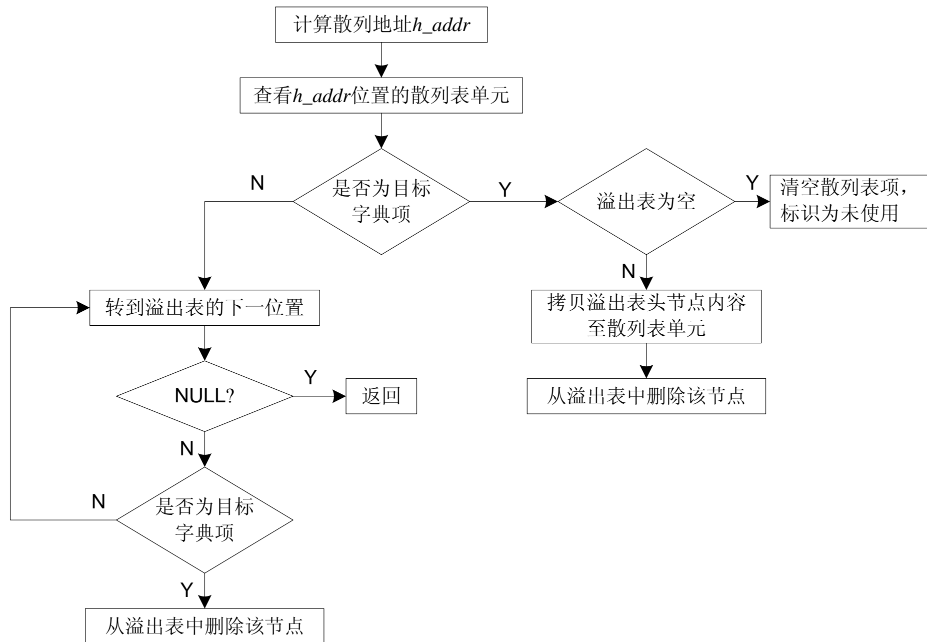


图 3-9 删除一个索引项

Fig. 3-9 Algorithm of Deleting an OD Entry

3.3.2.1 散列表大小的选择

这里散列表大小指的是散列表的地址空间大小 N 。对于 CANopen 对象字典来说，散列表应当能够在不引入大量冲突的的情况下，将对象字典放入散列表中。所以，N 应当足够大，以保证在一般情况下能够快速访问对象字典项。此外，N 也不应过大。溢出表算法已经保证了动态增加对象字典的容量以容纳新增数据项，因此过大的 N 将导致散列表空间浪费。从实现的角度，N 应当为 2 的幂，以简化代码。

CANopen 主站初始化过程中会建立对象字典项 1292 个（附录 1）。CANopen 主站每记录一个新从节点需要 14 个索引项用以配置该节点（附录 2）。此外，还有 PDO 相关的索引项和数据区。根据以上推断，一个小型的 CANopen 网络，CANopen 主站需要保存约 1500 个数据字典项。

本文选择散列表空间大小为 512，这样，平均每个散列表项只需要链接一个平均长度为 2 的溢出表。在没有引入大搜索延迟的前提下，合理地利用了空间。

3.3.2.2 散列公式的选择

如果散列表项的溢出表很短，溢出表搜索时间可以忽略不计，那么，散列表算法的搜索时间将主要决定于散列公式的计算。所以，应当合理选择散列公式，在保证高填充率的前提下，尽量缩短计算时间。

从运算选择上来说，微处理器适合计算 32 位以下的乘法运算，而不适合计算除法、开方等运算；适合采用截位进行对 2 的幂取余运算而对其它数的取余运算使用除法，计算缓慢。从算法的角度，散列公式应当尽量随机化从 *addr* 到 *h_addr* 的映射[28]。这样，CANopen 对象字典的散列公式应当采用式（3-1）的形式：

$$h_addr = ((index * r \gg s) + subIndex)\%N \tag{3-1}$$

其中 r 为随机因子，s 为位移因子，N 为散列表空间大小，取 512。采用加 *subindex* 的方法，在连续获得一个主索引的子索引的时候，可以省去散列公式的计算时间。整个公式包括一个乘法运算、一个加法、一个位移和一个截位（取余）运算，4 个时钟周期完成。

为了确定 s 和 r 的最佳取值，可以根据现有的初始对象字典进行仿真。由 s 和 r 的不同取值，可以确定一个取值平面。在取值平面上，对每一个点的平均搜索时间进行穷举，得到图 3-10。当 (r, s) 取 $(719,1)$ 的时候，获得最小搜索时间，填充率为 100%。

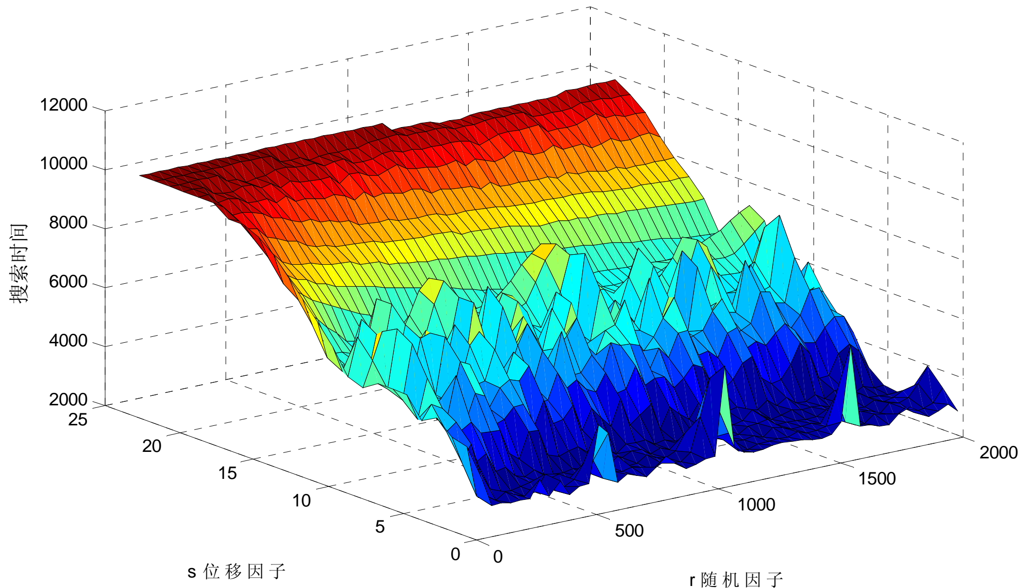


图 3-10 搜索时间仿真结果

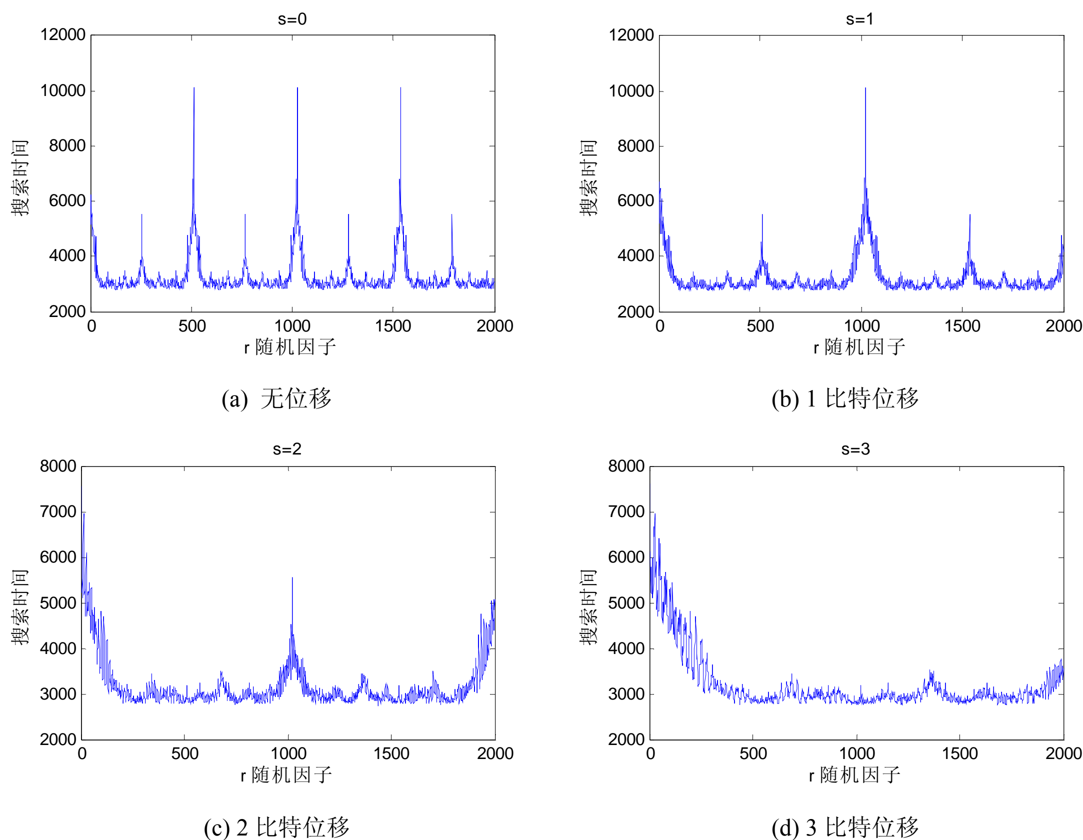
Fig. 3-10 Simulation of OD Searching Time

然而，是否最小点就一定是最佳点呢？最小点是根据对象字典的初始情况得到的，然而实际运行时的对象字典和初始对象字典会有很大区别，比如不同的节点号，不同的数据区，不同的 PDO 参数等等。因此，必须取一个和测试样本相关性较小的点来生成散列公式。

观察图 3-10，当无位移时，不同的 r 会造成搜索时间的大幅变化。随着 s 变大，r 对搜索时间的影响逐渐变小，但是搜索时间会随着 s 而整体增大。因此，s 能够减弱 r 对测试样本的依赖，然而会增加搜索时间的平均值。图 3-11 给出了当 s 取(0~3)时搜索平面的 4 个截面。从图中可见，当 s 较小时，搜索时间并没有显著增加，然而搜索时间相对 r 的变化率显著减小。本文选 s 取值为 2。进而在图 3-11(c)中得到搜索时间的最小点(1438,2)，代入到式（3-1）中得到最终的散列公式（3-2）。

$$h_addr = (index * 1438 \gg 2 + subIndex)\%512 \tag{3-2}$$

实验数据表明，该散列公式的填充率也为 100%。对搜索时间的穷举计算在附录 3 中给出。



(a) 无位移 (b) 1 比特位移

(c) 2 比特位移 (d) 3 比特位移

图 3-11 不同移位情况下的搜索时间

Fig. 3-11 Searching Time under Different Shift Number

3.3.3 性能分析及速度优化

散列表方法是数组和链表搜索的结合。对于散列表的搜索和数组相同，但是对溢出表则采用链表的搜索方式。下面将给出基于散列表的对象字典实现方法的平均搜索时间估计。

数组搜索由于下标的使用，为 $O(1)$算法。

链表只能单向搜索，如果链表中的节点为随机分布，那么平均搜索时间为 $O(n)$，其中 n 为链表的长度。

令 m 为对象字典项总数，N 为散列表地址范围。在散列表的填充率为 100%，$m \geq N$ 的情况下*，可以得到散列表的平均长度：

$$l = (m / N) - 1 \tag{3-3}$$

进而，散列表的平均搜索时间可表示为：

*当需保存的数据超出散列表地址范围几倍，并且散列公式合理，散列表的填充率一般接近 100%。

$$\bar{t}=\begin{cases}1 & ,(m<N)\\ \dfrac{N}{m}+\dfrac{m-N}{m}\bullet\left(\dfrac{\dfrac{m}{N}-1}{2}+1\right) & ,(m\geq N)\end{cases} \tag{3-4}$$

当 $m\geq N$ 时，$\dfrac{N}{m}$ 的对象字典项保存在散列表中，搜索时间为 1。$\dfrac{m-N}{m}$ 的对象字典项保存在溢出表中，搜索时间为 $\dfrac{l}{2}+1$。

化简式（3-4）可得：

$$\bar{t}=\begin{cases}1 & ,(m<N)\\ \dfrac{N^2+m^2}{2mN} & ,(m\geq N)\end{cases} \tag{3-5}$$

由式（3-5），散列表方法在 m 较大时仍然为 $O(n)$算法，然而该方法所使用的时间为链表方法的 N 分之一。也就是说，当 m 远大于 N 时，搜索时间和 N 成反比，较大的散列表宽度可减少平均搜索时间。图 3-12 同时给出了数组方法、链表方法和散列表方法的平均搜索时间。明显的，散列表方法在存在大量对象字典项的时候，搜索速度基本为链表方法的 1/256；即使在 10000 个对象字典项时，也仅比数组方法慢 10 倍以内。

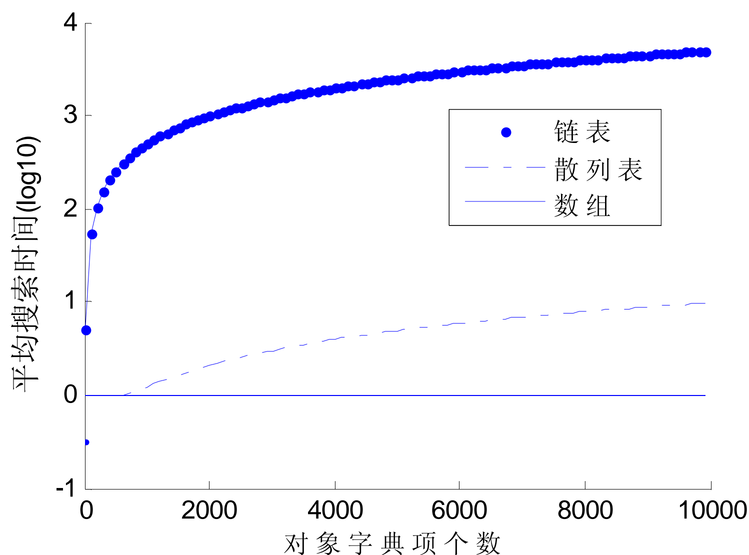


图 3-12 三种方法的搜索时间

Fig. 3-12 Searching Time of the Three Methods

当然，在存在大量对象字典项的情况下，比数组方法慢 10 倍也是缓慢的。所以需要对散列表方法进行进一步优化。

CANopen 主站并不是频繁地访问所有的对象字典项。只有 PDO 通讯的配置数据和一些常用数据区是被经常访问的，而一些器件信息和 SDO 的通讯配置往往只有在初始化过程中才被访问和更改。这便提供了一种加快散列表算法搜索时间的方法。

借鉴计算机 Cache 的思想，我们尝试将经常访问的对象字典项放在溢出表的较前位置，以减少常用对象字典项的访问时间。特别的，链表节点的添加和删除操作异常简单，该改进几乎不花费处理器时间*。改进后的搜索方法如图 3-13。

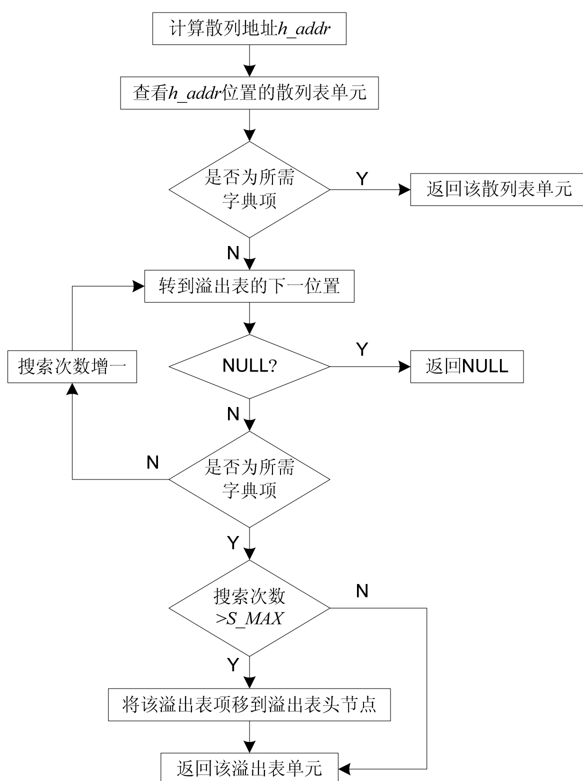


图 3-13 改进的对象字典项搜索算法

Fig. 3-13 Speeded OD Entry Searching Algorithm

*如果单从指令数量来看，改进算法的处理器运行时间反而会比原方法多。为了记录在散列表中的搜索次数，必须保存一个搜索计数器和当前节点的父节点（并没有在伪算法中标明）。并且对链表的删除和添加操作也需要至少 10 个指令周期和一个跳转，相反简单的散列表搜索，每节点只需要 4 条指令。但是，更主要的问题是，搜索算法实际上是在内存中不断读取不连续的数据，大量的时间会花费在内存总线的操作和 Cache 的缺页中断上，该时间远远比简单的指令执行时间要长。所以，减少搜索算法在溢出表中的搜索次数仍然能提高搜索速度[30]。

在改进算法中，增加了对溢出表搜索次数的计算。如果溢出表搜索次数大于阈值 *S_MAX*，那么，该对象字典项就被移到溢出表的首节点。这样，经常被访问的对象字典项会以较高的概率出现在溢出表的较前位置。这里并不是移到散列表中，因为内存拷贝一般会比指针操作操作慢很多。即使散列表中保存的是非常用对象字典项，内存拷贝的时间代价也足以让该对象字典项保持原位。

为了说明该方法的改进效果，假设对象字典项中有 1000 个常用对象字典项，*S_MAX* 取值为 3，访问常用对象字典项的概率为 80%，可以通过仿真得到改进之前和改进之后对象字典项的平均搜索时间和对象字典项总数之间的关系，如图 3-14。

当对象字典项总数较大的时候，改进算法的搜索时间明显小于原散列表算法。在对象字典项总数为 10000 时，改进算法的搜索时间基本为原算法的一半。实际上，由于 80%的对对象字典的访问都集中于 1000 个常用对象字典项，而阈值为 3 的时候，整个散列表和溢出表的前 3 项可容纳 2048 个对象字典项，大于 1000 的常用项个数。可推得，对于常用节点的访问，搜索时间小于 3 个溢出表节点的搜索时间。进一步可以得出，如果对常用对象字典项的访问概率趋向于 100%，对象字典项的平均搜索时间将小于 4（虚线表示）。如果阈值设置合理，使得任何时候访问常用对象字典项，溢出表的搜索次数都小于 *S_MAX*，那么，当常用对象字典项的访问概率趋向 100%时，搜索时间将小于 $(1+S_MAX)$ 个溢出表节点的搜索时间。

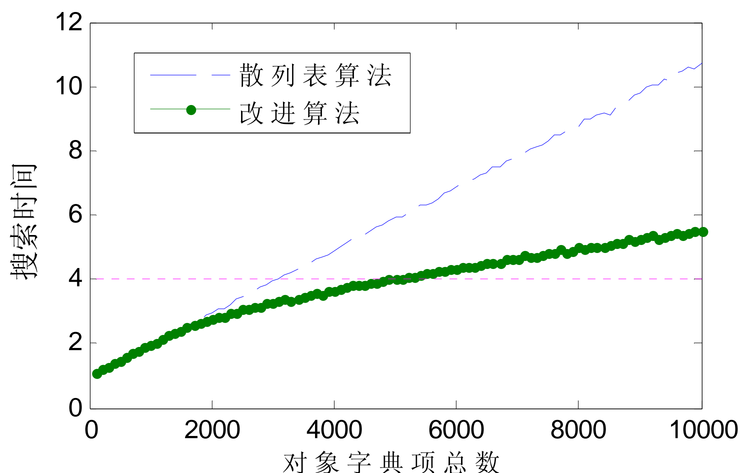


图 3-14 改进算法的性能分析

Fig. 3-14 Performance of the Improved Algorithm

这种改进的重要意义在于，只要对 CANopen 主站对对象字典的访问集中在常用对象字典项集合中，那么搜索时间将取决于对象字典中常用对象字典项的个数，而非整个对象字典的大小。然而一般 CANopen 节点的常用字典项集合大小是固定的，由节点的行为决定。在对象字典中添加无需频繁访问的其它细节信息将不会影响对象字典的实际访问速度。

3.4 本章小结

当前大多数 CANopen 协议栈采用数组变型的方式实现对象字典，本站提出了一种基于散列表实现对象字典的新方法。表 3-3 给出了这两种方法的性能比较。

表 3-3 改进的散列表方法与数组方法的性能比较

Table 3-3 Performance Comparison between Improved Hash Table and Array Algorithms

	改进的散列表方法	数组方法
读取时间	小于 $(1+S_MAX)$ 个溢出表节点的搜索时间	近似固定时间
添加和删除节点	容易	几乎不能
存储空间占用	数据字典项空间，每个字典项需 7 个字节保存索引号和链表指针	数据字典项空间和映射表空间

综上所述，散列表方法为对象字典提供了灵活的可扩展能力，保持了快速搜索特性，完全符合了 CANopen 主站对对象字典的可扩展、搜索快速和体积小的要求。

第四章 任务调度机的设计与分析

4.1 简介

本章将着重讲述针对混合动力汽车控制系统 CANopen 主站的性能要求和实际使用要求而设计的任务调度机算法、实现和性能。

4.2 任务调度方法的提出

4.2.1 任务调度的必要性

本论文将利用任务调度机实现 CANopen 主站。该任务调度机基于简单优先级的任务调度算法。使用该方法的原因来源于 CANopen 主站的运行特点和混合动力汽车控制系统的性能要求。

如第二章所述，CANopen 主站是 CANopen 网络的网络管理主站。根据功能要求，CANopen 主站需要具备实时运行、并行处理和灵活配置的能力。

实时运行：根据 CANopen 应用层协议规范[7]，PDO 通讯的同步周期计时单位为 1μs。并且，在 Renji V Chacko 等对于混合式动力汽车的系统要求描述中，明确提出了系统各模块之间的数据通讯间隔应为 500μs~1ms [31]。因此，CANopen 主站的时间辨识精度至少应达到 1μs，事件处理的延迟不应超过 500μs。

并行处理：作为 CANopen 网络的网络管理者，CANopen 主站需要同时与网络中的多个从节点进行 SDO、PDO 与 NMT 通讯。在网络启动阶段，CANopen 主站需要为网络中的每一个从节点建立一个线程以检查节点的配置信息和节点状态[22]。此外，PDO 通讯的一个基本思想就是通过为每个 PDO 报文分配不同的发送周期，来体现数据优先级和避免网络冲突。然而，正由于每个节点有不同的 PDO 发送周期，不可避免地在某一个同步周期，大量的从节点都处于 PDO 发送状态，导致突发性的报文传输。CANopen 主站很可能在这个周期不能完全处理，或是从节点未能竞争到同步窗口，而延迟发送 PDO 报文。这两种原因造成的数据滞后称为系统变量抖动[32]。尽管有一些算法试图通过动态配置 PDO 发

送周期减小系统变量抖动[33, 34]，但是它们都依赖于 CANopen 主站有足够的并行处理能力。

灵活配置：在 3.2 节已经叙述，CANopen 主站和从站不同，在设计阶段主站并不知道网络的具体拓扑，需要在运行时动态地加入和剔出从节点，开启或关闭 PDO 或者 SDO 通讯通道。这种灵活性不仅体现在对象字典的设计上，同时也体现在主站的整体结构上。最明显的，每加入一个新的从节点，主站需要开启一个新的 SDO 通道，意味着新的事件响应过程。

现行的实时嵌入式系统结构主要有两种：无操作系统的任务循环型结构和基于嵌入式操作系统的任务调度结构。

无操作系统的任务循环结构：开源的 CANopen 节点[11, 25, 26]一般都基于这种结构。以图 4-1 所示的 MicroChip CANopen 从站[26]为例，主函数在完成了硬件和 CANopen 初始化之后，就进入了任务循环。每一个任务由一个处理函数表示，依次被任务循环调用。如果任务循环中的每一个函数都执行得足够快，所有的任务则可被认为并行执行。然而，这种任务循环基本没有调度能力，各任务之间没有优先级的概念，所以实时性高的事件和一般事件一样都必须等待整个任务循环运行到相应的处理函数才能获得响应。如果一个任务发生阻塞，执行时间过长，将导致所有事件的响应时间加大。另外，任务循环不具备运行时扩展能力，所有的任务在编译时已经确定，所以基于任务循环结构的 CANopen 协议栈将不能动态添加 PDO 或 SDO 通讯对象，基于此实现的主站则更不可能具备动态添加新节点的能力。

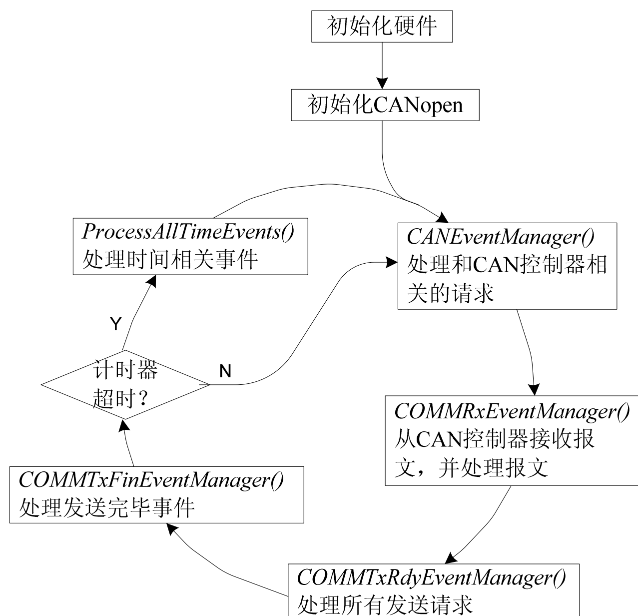


图 4-1 MicroChip CANopen 的任务循环结构

Fig. 4-1 Process Loop of MicroChip CANopen Stack

基于嵌入式操作系统的任务调度结构：商业的 CANopen 主站 Beckhoff TwinCAT [35, 36]基于普通的 Windows 操作系统，开源 CANopen 主站 OCERA [27]基于 RTLinux 操作系统。在操作系统动态内存管理和动态线程管理的帮助下，CANopen 主站可以很好地支持动态配置。然而，并非所有的操作系统都能达到 CANopen 网络所要求的实时性。表 4-1 列出了当前使用的几种操作系统的事件响应能力和时钟精确度。

表 4-1 几种操作系统的实时性能比较

Table 4-1 Real-Time Performance of Several Operation Systems

操作系统	测试平台主频	事件响应延迟	时钟精度
WindowsCE	133MHz	>34μs[37]	>5ms
Linux	1.8GHz	>200μs[38]	>5ms
RTAI μCLinux	120MHz	>25μs[39]	硬件计时器
μC/OS-II	67.5MHz	3.8μs[40]	硬件计时器

首先，Linux 系统的实时性并不满足要求，在 1.8GHz 的主频下，事件响应时间仍然大于 200μs。WindowsCE 虽然能够满足实时性要求，然而操作系统只提供精度小于 5ms 的系统时钟，显然不能满足 CANopen 同步周期以μs 为单位的计时要求。RTAI μCLinux 和μC/OS-II 系统能满足所有的时间要求。其中 RTAI μCLinux 在 Linux 内核与硬件之间架设了一个 RTAI 内核，该结构一定程度上解决了 Liunx 内核的事件响应速度问题，但是也为编程和系统移植带来困难。μC/OS-II 在系统移植性和实时性上完全满足要求，但是它最多支持 64 个线程[41]，限制了动态可配置能力；此外，μC/OS-II 提供的内存分配函数不支持完全的动态内存分配[41]，对实现基于散列表的对象字典造成困难。另外一个显著的问题是操作系统的价格，例如 WindowsCE 和 RTLinux 系统，它们并不是免费的，μC/OS-II 的一些功能也是需要付费的，这些都给 CANopen 主站的实现造成困难。

反观 CANopen 主站对实现的要求实际上只存在于实时性、多线程和动态内存分配这三点上。因此，本文提出一种简单的、不依赖于操作系统的、完全基于标准 C 的任务调度机模型。由于该任务调度机不依赖于操作系统，在控制器上直接运行时反而能获得较高的实时特性，同时该调度机也提供了适合于 CANopen 主站的多线程调度特性，配合由 C 编写的动态内存分配函数，则能满足 CANopen 主站的所有要求。并且，由于整个调度机用标准 C 编写，可以编译并运行在所有的支持 C 编译的硬件平台之上，具有最强的移植能力。即使目

标平台是一个有操作系统环境，该调度机可作为操作系统的一个实时线程，仍然能够高效运行[42]。

4.2.2 调度算法的选择

实时系统[43]的调度算法主要分为两种：抢占式和非抢占式[44]。

一般的实时操作系统都采用抢占式的任务调度算法，例如 WindowsCE、RTLinux 和μC/OS-II。该任务调度算法的主要特点是当高优先级的事件到达时，操作系统能够中断低优先级的线程，并执行高优先级事件的响应线程。因此，抢占式的操作系统能够保证事件的响应时间小于一个固定的最坏事件响应时间，一般使用于需要实现硬实时性能的操作系统当中。其主要问题是需要硬件和汇编的支持，标准 C 难以实现抢占式的调度算法。此外，由于在发生抢占的时候需要保存被中断线程的上下文，不适当的抢占还会导致死锁，需要调度机去检测死锁并解除死锁状态。抢占式的调度算法一般比较复杂。

非抢占式的任务调度算法一般运用于软实时的操作系统。调度算法必须等待当前的线程进入睡眠、被阻塞或执行完毕之后才能调度另外一个合适的线程继续执行。由于调度机不能中断一个正在运行的线程，因此非抢占式的调度机不需要和抢占式的调度机一样保存被中断线程的上下文。更重要的，这种机制也从根本上防止了死锁的发生。

相比较而言，CANopen 网络通讯属于软实时系统。PDO 等实时数据对象允许延迟的发生，因而小范围的延迟不会对控制系统有较大的损害。由于抢占式的调度机需要汇编代码的支持，和可移植性的目标相违背，本文选择非抢占式的任务调度算法[45]。

关于非抢占式的任务调度算法，其主要问题是在保证处理器较高有效载荷的前提下实现接近硬实时的性能。换句话说，就是尽量减少处理器用在调度算法上的时间，并且让实时事件的响应时间最小。由于任务调度算法不能在线程的运行过程中中断该线程，实现响应时间最小的办法就是以最少的时间让最需执行的任务获得执行。当前使用的最多的为最早截止时间优先（EDF）算法[46]。

EDF 算法的主要思想是在每一个调度周期，计算所有可运行任务的截止时间，选择截止时间最短的任务运行。在该算法下，优先级高的事件由于截止时间短，而较容易被响应，最终提高系统的实时处理能力。对一般的操作系统而言，每一个线程没有明显的截止时间，线程的运行时间也不固定，所以需要通过以前的运行数据估计线程的截止时间和运行时间。该估计需要计算大量的运行时间统计信息[47]。

然而，实现 EDF 对于 CANopen 主站来说，却比较容易。CANopen 应用层协议根据通讯的优先级将报文按表 2-1 的方式分成了 6 大类。每一类报文代表

一种通讯事件，其优先级由报文的 COB-ID 显式表示，直接反映了该通讯事件的允许响应延迟。因此，实现 EDF 方式的非抢占式任务调度机算法，实际上就是一种严格按照 COB-ID 定义的固定优先级的高优先级优先调度算法。优先级本身就代表了截止时间，实现 EDF 算法不需要对截止时间进行估计。实际上，运行时间也不用估计。运行时间一般作为参数参与线程的优先级运算，固定优先级则不用重新计算任务优先级*。

综上所述，由于 CANopen 特殊的报文定义方式，通讯事件本身天然地符合了线程优先级的定义方式。使用非抢占式的任务调度机算法，可以较容易地实现 EDF 算法，获得较短的响应时间。并且，所有的算法可以使用标准 C 实现而无需汇编代码的帮助，使得该调度机算法可以移植于任何支持标准 C 编译器的嵌入式平台。

4.3 任务调度机的实现

4.3.1 任务的抽象

我们定义任务为线程的执行对象，也是线程的表现形式。一个任务必须能够以一种具体的、可操作的数据对象形式，保存一个特定功能的执行过程，以及该过程运行所需要的私有数据空间。只有以这种具体的形式，一个任务才能被保存在队列当中，实现任务调度。

图 4-2 给出了任务的数据结构定义。在该数据结构中，*runPrio* 保存了该任务的优先级。根据 CANopen 报文的分类，任务的优先级分配如表 4-2。报文收发任务为最高优先级 0。同步帧直接影响到 PDO 的正常通讯，所以同步生成任务也为最高优先级。其次依次为 PDO 的报文分发和处理任务，SDO 的报文分发和处理任务，NMT 节点状态报文的分发和状态监视任务。最低优先级 8 为用户的配置任务。

---

*事实上，在极端情况下，完全固定优先级算法确实会使得低优先级的线程饿死。这时便需要动态优先级。提高长时等待线程的优先级使其获得运行的权利。该方法作为可选方法，已经在实际的 CANopen 主站调度机中实现。

```
typedef struct _TaskObj {
    char runPrio;                  // 任务优先级
    char state;                    // 任务当前状态
    // 任务处理函数
    char (*pFun) (struct _TaskObj *);
    int Argu[10];                  // 任务的私有数据空间
    int timeHigh;                  // 时间
    int timeLow;                   // 时间
    struct _TaskObj * pNext;       // 任务队列指针
    struct _TaskObj * pPre;        // 任务队列指针
} TaskObj, *pTaskObj;
```

图 4-2 任务的数据结构

Fig. 4-2 Data Structure of Task

表 4-2 任务的优先级分配

Table 4-2 Priority Allocation of Tasks

优先级	任务描述	优先级	任务描述
0	报文接收和发送任务	5	NMT 节点状态报文分发任务
0	同步报文生成任务	5	时间保文发送
1	PDO 报文分发任务	6	节点保护和心跳报文监视任务
2	PDO 报文发送任务	7	从节点启动检查任务
3	SDO 报文分发任务	7	从节点 PDO 配置任务
4	SDO 报文处理任务	8	用户请求

state 保存了任务的当前队列状态。当任务在等待任务队列时为 0，进入运行任务队列时为 1。其主要目的是防止任务的重复触发。关于任务队列会在下一节仔细描述。

pFun 为任务的执行函数。该函数负责完成任务的特定功能，实际上也是任务的功能体现。整个任务结构体的最根本作用就是将任务的执行函数封装成一个独立的数据结构以便排队。*pFun* 拥有一个 40 字节大小的私有数据空间 *Argu*。这里我们将数据空间直接在任务结构体中显式表示，省去了进一步抽象线程私有栈空间的必要性。

pFun 的输入参数只有该任务结构体自身的指针。这样有两个好处：一，统一了任务函数的定义；二，任务函数可以方便地通过该输入参数访问任务结构体，特别是私有数据空间。如果一个任务需要一些输入变量和初始值，由于私有数据空间的存在，输入变量和初始值都可以在任务初始化时保存在数据空间内。C 语言的 *main* 函数的形参 *argc* 和 *argv* 也有同样的功能。

pFun 的返回值包括 TASK_OK、TASK_WAIT、TASK_RUN 和 TASK_ERR。其中 TASK_OK 表示该任务执行完毕，可以释放该任务空间。TASK_WAIT 表示任务需要等待一个事件或者等待一段时间后继续执行，应当进入等待任务队列。TASK_RUN 说明任务还未执行完毕，但是主动退出，使得调度机获得一个重新调度的机会，任务进入运行任务队列。TASK_ERR 说明任务运行出现严重错误，调度机会因为该错误而停止并退出整个系统。

timeHigh 和 *timeLow* 合起来表示一个 64 比特的微秒级时钟。由于 32 比特的时钟会在 71.6 分钟后溢出，所以这里使用两个 32 比特的整数表示绝对时间。当任务处于等待任务队列时，该时间表示该任务下一次执行的预期开始时间。当实际时间大于该时间时，说明该任务应当获得执行，即进入运行任务队列；当任务处于运行任务队列时，该时间保存任务进入运行任务队列的时间，和实际时间的差即为该任务在运行任务队列中的等待时间。由于每一个任务单独地保存了自己的时间，省去了为每个任务单独建立计时器的必要，整个系统只需维护一个实时的 64 比特微秒级时钟即可满足任务调度机的所有时间要求。

pNext 和 *pPre* 为链表指针。任务会以链表的方式放入等待任务队列和运行任务队列，均为双向链表的结构。

4.3.2 调度算法

如图 4-3，调度机由等待任务队列、执行任务队列和调度算法组成。等待任务队列中保存所有暂时不满足执行条件的任务，执行任务队列则保存已经满足执行条件，但是还没有被处理器执行的任务。

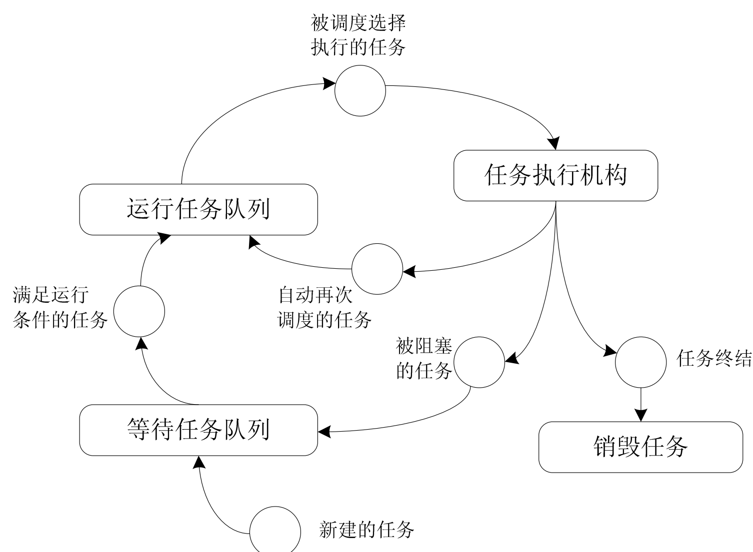


图 4-3 任务调度机的结构

Fig. 4-3 Structure of Task Scheduler

新建的任务首先被放入等待任务队列。当一个任务执行完毕，调度算法执行。调度算法会遍历等待任务队列，将所有满足运行条件的任务转移到运行任务队列。然后在运行队列中选择拥有最高优先级的任务执行。任务执行完毕后，可返回等待任务队列、运行任务队列，或者被销毁。以伪代码形式表示的调度算法由图 4-4 给出。

在伪算法中，*RQ* 和 *WQ* 分别代表运行任务队列和等待任务队列。*MoveTaskToRun()*方法扫描等待任务队列，并将所有满足执行条件的任务置入运行任务队列。*SelectTaskForProc()*方法从运行任务队列中选出优先级最高的任务。根据任务的执行结果 *rV*，任务将返回运行任务队列、等待任务队列或被销毁。整个任务调度算法为一个死循环，只有当出现异常错误时才会退出。

```
void scheduler(RunQueue RQ, TaskQueueHeader WQ)
{
   while(1) {
      MoveTaskToRun(RQ,WQ);             // 更新可执行任务
      if(SelectTaskForProc(&pTRun, RQ))// 选择任务
         continue;                      // 无任务可执行
      rV = (pTRun->pFun)(pTRun);        // 执行任务
      switch(rV) {
         case TASK_OK:                  // 执行完毕，释放空间
            释放任务空间; break;
         case TASK_RUN:                 // 主动放弃执行，重新调度
            将任务放入运行任务队列; break;
         case TASK_WAIT:                // 等待条件，返回等待
            将任务放入等待任务队列; break;
         default:                       // 异常退出
            return;
      }
   }
}
```

图 4-4 调度算法伪算法

Fig. 4-4 Pseudo-code for Schedule Algorithm

明显的，任务调度机算法应当尽量减少对处理器的占用，以提高有效载荷。为了减少 *SelectTaskForProc()*方法的执行时间，本文采用类似于 Linux 2.6 内核的运行队列结构[45, 48, 49]（图 4-5）。整个运行队列由 9 条优先级队列组成，每个优先级队列有一个非空标志位。根据图 4-6 所示的算法，算法从最高优先级队列开始寻找，直到找到一个非空队列，取出头节点，该节点即为下一个执行任务。由于优先级队列共 9 条，该算法的最大执行时间固定，为 $O(1)$算法[48]。

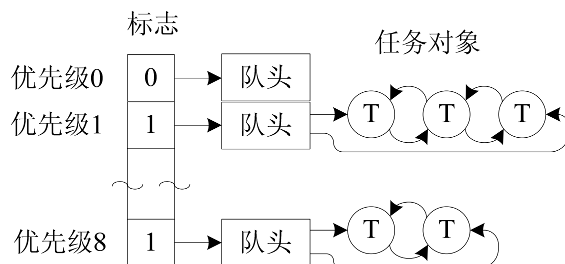


图 4-5 运行队列结构

Fig. 4-5 Structure of Runnable Queue

等待任务队列的简单实现方式为无序双向链表。该实现的好处是添加任务简单。由于双向链表无序，所以添加一个新的任务到等待任务队列只需将该任务添加至链表的尾部，为 $O(1)$算法。但是，调度过程中 *MoveTaskToRun()*方法必须遍历整个等待任务队列，将所有满足运行条件的任务转入运行任务队列，执行时间和等待任务队列的长度 n 成正比，为 $O(n)$算法。因此，当等待任务数较大时，调度算法执行时间较长。

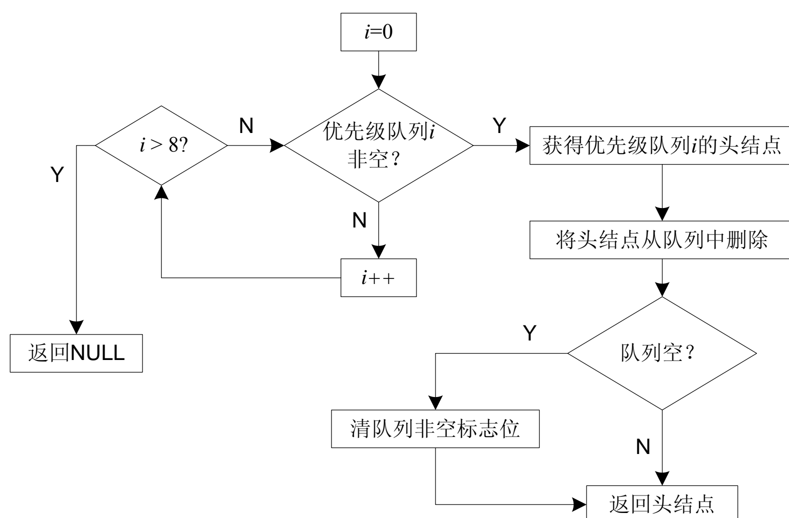


图 4-6 下一执行任务的寻找方法

Fig. 4-6 Algorithm for the Next Runnable Task

一种改进的方案是有序化等待任务队列。如果将等待任务队列按照时间从小到大排序，那么 *MoveTaskToRun()*方法就只需要搜索到第一个不满足时间运行条件的任务时就可以停止搜索，剩下的任务都不可能满足时间要求。但是，为了保持等待队列的有序化，将任务添加到等待任务队列的操作不再是简单的插入队尾操作，而需搜索等待任务队列，寻找合适的插入位置，算法复杂度退化为 $O(n)$。好在任务不仅由时间触发，也可由事件触发。如果一个任务只由事件

触发，其下一次执行时间为时间最大值，因而可直接插入等待任务队列队尾，仍然为$O(1)$算法。

比较两种方法的整体时间复杂度：

假设在一个调度周期内有 n 个任务加入已存在 m 个任务的等待任务队列。包含时间约束的任务占任务总数的比为 e，等待任务队列中在下一调度时刻满足时间要求的任务与等待任务队列的任务总数之比为 r。定义搜索一个任务节点的耗时为 1。那么，无序等待任务队列的总耗时 t 可表示为：

$$t = m + 2n \tag{4-1}$$

其中添加 n 个任务至等待任务队列花费时间为 n，遍历等待任务队列花费时间为$(m+n)$。

对于有序等待任务队列，将一个任务放入含有 m 个任务的等待任务队列耗时为 $t_{insetr,m}$：

$$t_{insert,m} = (1-e) + e \cdot \frac{m \cdot e + 1}{2} \tag{4-2}$$

对于每一个新任务，该任务有$(1-e)$的可能仅由事件触发，e 的可能包含时间触发条件。如果该任务仅事件触发，插入等待任务队列的耗时为 1，为式（4-2）中的第一项。如果该任务包含时间触发条件，则需要在等待任务队列中寻找一个合适位置，寻找的最大长度为所有的等待任务队列中包含时间触发信息的任务总数加 1，即为$m \cdot e + 1$。由此得到式（4-2）中的第二项。

将所有 n 个任务的插入时间相加，可以得到一个调度周期内，CANopen 用在插入等待任务队列操作上的总耗时t_{insert}：

$$t_{insert} = \sum_{i=m}^{m+n-1} [(1-e) + e \cdot \frac{ie+1}{2}] \tag{4-3}$$

化简式（4-3）得到式（4-4）。

$$t_{insert} = \frac{e^2 n^2 + (2me^2 - e^2 - 2e + 4)n}{4} \tag{4-4}$$

在调度时刻调度机需要部分遍历等待任务队列直至找到第一个不满足时间触发条件的任务，因而整个搜索过程的耗时 t_{search} 可表示为：

$$t_{search} = (m+n)r + 1 \tag{4-5}$$

合并（4-4）和（4-5）得到有序等待任务队列的总耗时t'为：

$$t' = \frac{e^2n^2 + (2me^2 - e^2 - 4e + 4r + 4)n + 4mr + 4}{4} \tag{4-6}$$

在式（4-6）中，m 和 n 为变量；e 和 r 为常量，由 CANopen 的运行特性决定。一般情况下，PDO 和 SDO 报文分发任务、PDO 报文处理任务和用户任务仅由事件触发，所以大部分的任务都采用事件触发方式。为了方便比较，假设 e 为 30%。如果处理器执行任务的速度足够快，调度周期则较小，在一个调度周期内从不满足时间条件到满足条件的任务数则较少，r 可认为趋近于 0。这里假设 r 为 5%。

根据以上假设，图 4-7 反映了在不同 m 的情况下，两种等待任务队列方法的耗时比较。根据图 4-7(a,b)无论当前等待任务队列的任务数为 20 还是 100，有序等待任务队列的耗时都比无序等待任务队列少。根据图 4-7(a,c)，由时间约束的任务与任务总数之比 e 对有序等待任务队列的耗时有很大影响。其主要原因是 e 的增大直接导致在保存新任务至等待任务队列的过程中，平均搜索长度加大。相比之下，由图 4-7(c,d)可得，r 对耗时的影响则较小。由于 r 只决定调度时刻对等待任务队列的搜索长度，而有序等待任务队列的主要耗时在于新任务的插入。不过，r 间接决定了 m 的大小。当 r 过小时，特别当 $mr < n$ 的时候，整个系统的新任务产生速度将超过任务的执行速度，m 将会不断变大。根据图 4-7(a,b)，耗时也会随 m 比例增大。

根据图 4-7(c)，当 e 过大并且调度周期内的新任务数 n 过大时，有序等待任务队列方法将产生更大的耗时。然而，一个实时系统不允许在一个任务执行时间内平均产生大于 1 个的新任务，这将导致任务队列的不断增长。因此，只要在所有的参数允许范围内，确保 $n<1$，有序等待任务队列的耗时较小。本文采用有序的等待任务队列。

就调度算法而言，关于等待任务队列的操作耗时为 t_{search}，所以，整个调度算法的总耗时 $t_{schedule}$ 可表示为：

$$t_{schedule} = (m+n)r + 1 + C \tag{4-7}$$

式中 C 代表消耗 $O(1)$时间的 *SelectTaskForProc()*方法。由于 $n<1$，可以得到结论，本文所提出的任务调度机为 $O(m)$算法，其中 m 为等待任务队列的长度。

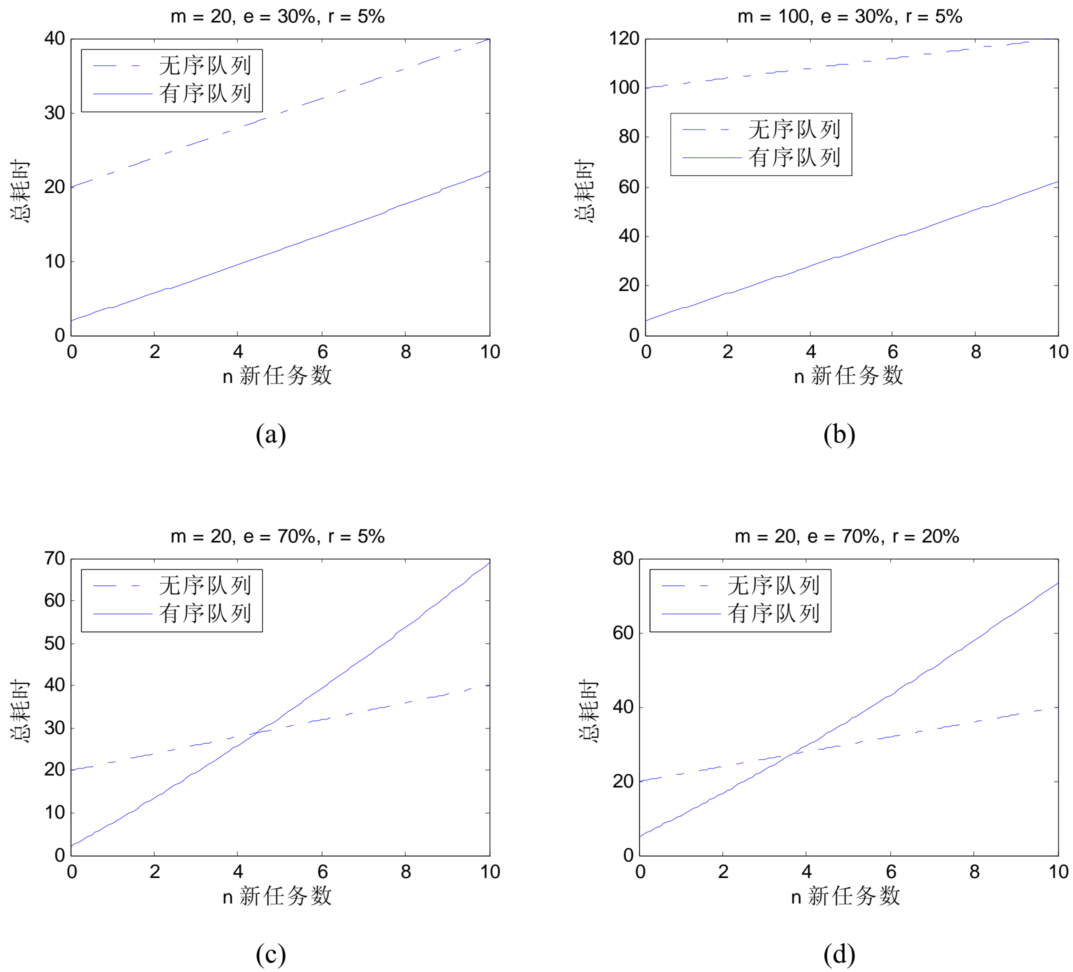


图 4-7 无序和有序等待任务队列耗时比较

Fig. 4-7 Time Consumption of Random and Ordered Queue of Waiting Task

4.4 性能分析

本文采用基于 Pentium-4 1.6G 中央处理器的 Windows XP Professional 操作系统平台和基于 ARM7 LPC2294 （11MHz）的无操作系统平台两种环境测试任务调度机的运行性能。在两个平台上，测试代码基本相同，都包含两种基本的任务：TaskGenerator 和 TaskTestRountine。

TaskGenerator 为常置任务，负责生成随机执行时间、随机优先级的 TaskTestRountine 任务，并将其放入等待运行队列。TaskTestRountine 任务执行一个固定耗时运算，并随机决定任务的返回值和下一次执行时间。基于这两个任务，任务调度机将不间断地执行。

在 Windows 平台上的运行结果如图 4-8：

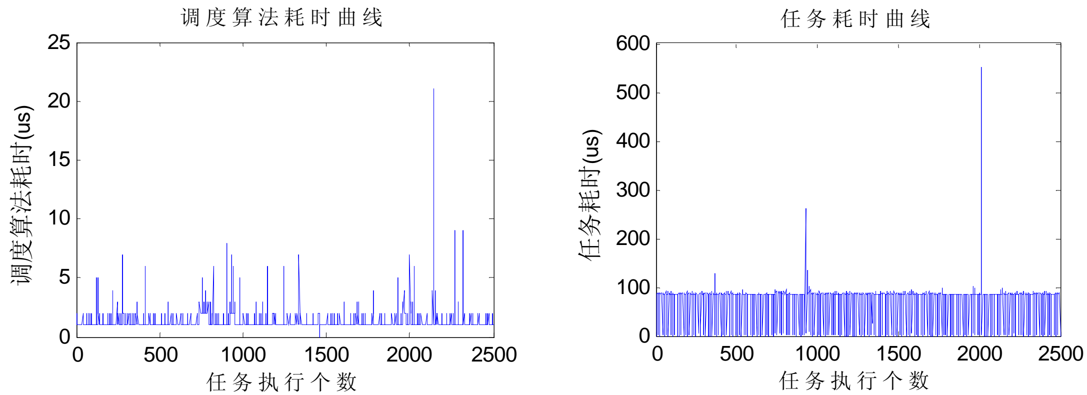


图 4-8 Windows 平台调度机测试结果

Fig. 4-8 Test Result of Scheduler on Windows

可见，调度算法的耗时在 1μs~2μs 之间。但是，在一些离散点上调度算法的耗时会超过 5μs，甚至达到 20μs。该现象同样也出现在任务的耗时曲线上。由于整个任务调度机作为 Windows 操作系统的一个线程，必然受到 Windows 操作系统的调度算法影响，因此，该现象的产生是由于 Windows 的调度算法打断了 CANopen 调度机的执行。基于这样的测试数据算出的平均调度时间较其真实值偏大。实验结果表明，在 Windows 操作系统上运行，调度算法的平均调度时间为 2.28μs[42]。

基于 ARM 的无操作系统平台，实现调度机的前提是动态内存分配。由于没有操作系统的支持，CANopen 主站必须自己实现动态内存分配机制。关于简单的动态内存分配的实现，附录 4 提供了 malloc 和 free 两个函数的伪代码。

图 4-9 给出了 ARM 平台的测试结果。和 Windows 平台的运行结果相比，ARM 平台的运行结果明显没有大范围的无规律抖动。这也间接证明了在 Windows 平台上，CANopen 调度机的运行受到操作系统调度机的影响。

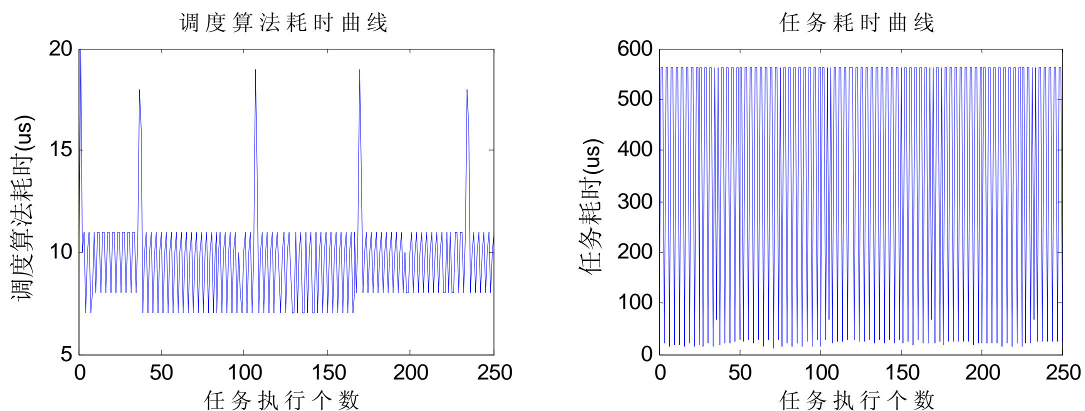


图 4-9 ARM 平台调度机测试结果

Fig. 4-9 Test Result of Scheduler on ARM

从图 4-9(a)可以看出，调度算法的耗时一般在 10μs 左右，但是差不多周期性地会产生一个大的耗时。其主要原因是上一个运行任务 TaskGenerator 产生了大量的新任务，使得等待任务队列长度突然加长。根据调度算法的计算复杂度，调度算法耗时也比例增加。实验结果表明，在 ARM 平台上，调度算法的平均调度时间为 12.63μs[42]。

该时间明显比 Windows 平台的平均调度时间大，主要受到两方面的影响：首先 ARM 平台的主频较低，只有 11MHz；另外，ARM 平台没有缓存机制，内存总线操作较慢，制约了任务搜索算法的执行。12.63μs 还只是片内 RAM 运行的结果，如果使用片外内存作为任务结构体的存储空间，平均调度时间将达到 80μs 以上。

4.5 本章小结

综上所述，调度机在 Windows 平台和 ARM 平台的调度时间分别为 2.28μs 和 12.63μs。调度时间收到两个因素的影响，等待任务队列的长度，和单次内存读取耗时。如果与表 4-1 中其它操作系统的事件响应延时比较，考虑到实验平台的主频为 11MHz，12.63μs 的调度时间基本和 μC/OS-II 操作系统的响应时间相当。因此，该调度机在保证了可移植性的同时完全能够满足 CANopen 主站的实时性要求。

如果调度机的性能需要改进，可以从以下两个方向考虑：

1. 调度机的性能主要受到内存访问速度慢的限制。如果能够将所有的任务结构体放入一个合适大小的连续内存区域，同时具有高速缓存机制，该内存连续区域就能够一次性装入高速缓存，等待任务队列搜索过程中则不用产生缺页中断，大量减少搜索时间。在嵌入式的微控制器平台，该内存区域可以映射到片内内存，减少外部内存总线慢速访问对调度算法的影响。
2. 进一步修改任务等待队列的链表结构。一种方法是使用 binary heap 实现等待任务队列。根据 binary heap 的基本操作特性[29, 50]，添加新节点的计算复杂度为 $O(\log_2 m)$，m 为等待队列的节点总数。在 binary heap 中，最小节点即根节点，删除根节点的计算复杂度也为 $O(\log_2 m)$。当 m 较大时，该方法将获得更短的调度算法耗时。

第五章 基于调度机的 CANopen 主站协议栈设计

5.1 简介

本章将首先给出基于调度机模型而建立的 CANopen 主站协议栈的软件结构。然后各小节将依次分析各功能模块的作用和实现方法。

5.2 协议栈的整体结构

基于第四章所述的可移植的非抢占式任务调度机，CANopen 主站协议栈结构如图 5-1。

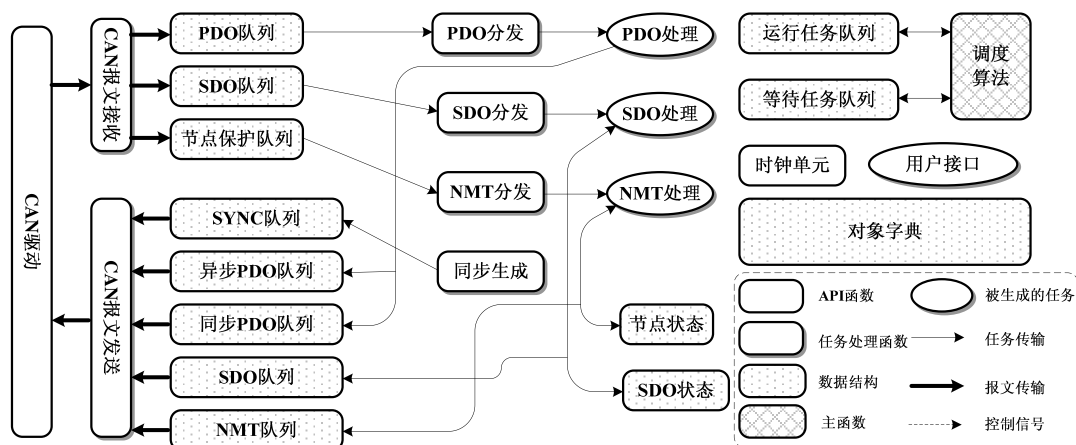


图 5-1 CANopen 协议栈结构

Fig. 5-1 Structure of CANopen Stack

整个 CANopen 主站协议栈由调度机、任务队列、报文队列和 CAN 驱动构成。调度机负责管理任务队列并选择合适的任务运行。在任务队列中，PDO 分发、SDO 分发、NMT 分发、同步生成、CAN 报文接收和 CAN 报文发送这 6 个任务为常置任务，在调度机初始化时启动并一直执行。CAN 驱动接收并发送 CAN 报文，各种常置任务根据接收到的报文产生新的响应任务处理报文。报文

队列则保存未处理的报文，为任务之间通讯的媒介之一。用户通过新建任务和读写对象字典的方式和协议栈进行交互。

后文将根据 CANopen 主站的各项功能要求分别分析 CANopen 主站协议栈的具体设计。

5.3 驱动与报文队列

一般 CAN 控制器提供两种 CAN 报文接收方式：轮询和中断。轮询方式为应用程序主动访问 CAN 控制器的接收缓冲区查询报文，中断则为 CAN 控制器通过中断主动通知应用程序新报文的到达。

尽管轮询方式编程简单，并且从本质上符合非抢占式调度机的运行机制，但是该方式对报文的响应速度缓慢，和 CANopen 主站的实时要求违背。因此，CANopen 主站协议栈使用中断的报文接收方式。

中断方式的最大问题是数据同步。如果将中断服务程序也看作一种线程，那么该线程具有最高的优先级，能够打断 CANopen 调度机的执行。中断服务程序又必须将接收到的报文传递给由 CANopen 调度机控制的各报文处理任务。所以，中断服务程序和调度机之间存在数据同步问题。

为了解决该问题，中断服务程序和报文处理任务之间使用先入先出队列（FIFO）作为通讯媒介。中断服务程序接收到报文之后，将报文保存入 FIFO，更改写指针。CANopen 报文处理任务检查 FIFO 的读写指针以确定新报文位置，读取新报文并更改读指针。由于中断服务程序只更改写指针，而报文处理任务只修改读指针，读写指针一起可以保证中断服务程序和报文处理任务不会对 FIFO 的同一位置进行更改，则解决了同步问题。唯一的代价是 FIFO 的处理过程中需要填充和拷贝报文内容，相比链表方式，FIFO 的报文处理速度稍慢。CAN 报文的发送，同理采用 FIFO 方式。此外，中断服务程序将报文放入 FIFO 之后，还应直接触发相应的报文处理任务运行，也就是将报文处理任务从等待任务队列放入运行队列[*]。

另外一个问题，由于中断服务程序可能修改任务队列，如果中断服务程序正好打断调度机的处理过程，则可能破坏等待任务队列和运行任务队列，应对这两个核心任务队列实施保护。如果将这两个核心任务队列看成一个共享资源，

---

[*] 这里为事件触发方式。执行触发的任务拥有被触发任务的结构体指针，按照 4.3.1 小节关于 state 的叙述，通过 state 可以知道该任务是否已经被触发。如果 state 为 0，说明该任务处于等待任务队列，没有被触发，则执行触发的任务将该任务移出等待任务队列，并放入运行任务队列，同时更改 state 为 1。

那么实施保护的根本方法就是实现对共享数据的原子操作[48, 51]。为了更好地管理核心任务队列，所有的操作必须由表 5-1 所示的几个函数完成。

在这些函数中，所有对于核心任务队列的操作都必须先获得锁，在操作完毕之后释放锁。不同的平台上获得锁的操作也不相同，所以该函数由具体平台决定。在无操作系统的嵌入式平台上，全局只有中断和调度机抢占该锁。如果中断服务程序在获得锁的过程中失败，那么中断将不能退出。因此，在嵌入式平台上的加锁过程即关中断操作。

表 5-1 对核心任务队列的处理函数

Table 5-1 Functions Handling the Kernel Task Queues

函数名	调用者	说明
MoveTaskToRun()	调度机	将所有满足运行条件的任务从等待任务队列移至运行任务队列
NewTask()	任何任务	新建一个任务结构体*
AddTaskToWait()	调度机	将任务放入等待任务队列
SelectTaskForProc()	调度机	在运行任务队列中选择一个优先级最高的任务
DeleteProcessedTask()	调度机	删除一个任务结构体
ReInsertTask()	调度机	将一个任务放入运行任务队列
WakeTask()	任何任务	将一个特定任务从等待任务队列移至运行任务队列

从数据保护问题的解决也可看出，一些特殊操作必须获得特定平台的支持，和 CANopen 主站的可移植性要求相矛盾。为了提供更好的可移植性，关于 CAN 报文的接收和发送被分为两个部分：中断服务程序和报文收发任务。其中中断服务程序为抽象层，具体实现由平台确定；报文收发任务为 CANopen 调度机的常置任务，类似 linux 中的中断后半段（interruption bottom half）[48,51]，按照正常的调度过程执行；两个部分使用 FIFO 进行通讯，收发任务的实现和平台无关。

* 任务结构体在通常情况下并不由动态内存分配函数动态分配。动态内存分配函数执行较慢，而新建任务的操作很频繁，因而被释放的任务结构体会以链表的形式保存在待分配任务结构体链表当中。新建任务结构体的工作实际上是在待分配任务结构体链表当中选择一个节点的操作。只有当该链表为空时才会重新动态分配新任务结构体。由于对该链表的操作也可能被中断，同时中断服务程序也可能需要新建任务，导致该队列也需保护。

CAN 报文接收任务在 CAN 报文到达后由中断服务程序唤醒。该任务从 FIFO 中获得 CAN 报文后，将报文按照 COB-ID 放入不同的 CANopen 报文队列。如 2.3 节所述，CANopen 报文按照 COB-ID 的功能号分类，具有严格的优先级。根据该优先级，低优先级的报文只有当高优先级报文全部处理之后才能得到响应。为了实现这种优先级特性，接收报文会被 CANopen 报文接收任务送入接收 PDO、接收 SDO 和接收 NMT 等三个接收报文队列并触发相应报文分发任务执行。紧急报文直接由接收报文任务处理并上报应用程序。

同理，CANopen 报文发送也按照 COB-ID 分为不同优先级。CANopen 主站协议栈按照优先级从高到低分为同步报文、异步 PDO、同步 PDO 报文、发送 SDO 和发送 NMT 等五个报文队列。低优先级的报文只有当上一个优先级的报文队列为空时，才能发送。PDO 报文分为同步和事件类型（异步）两种。由于同步 PDO 报文受到 PDO 发送窗口的限制，在发送窗口外同步 PDO 将不能发送，因此同步和异步 PDO 被分为两个发送报文队列处理。

基于以上的讨论，图 5-2 给出了 CANopen 主站协议栈 CANopen 报文收发的软件结构[45]。

(a) 报文发送方向　　(b) 报文接收方向

图 5-2 CANopen 主站协议栈报文收发软件结构

Fig. 5-2 Software Structure of CAN Message Transmission and Receiving in CANopen Master Stack

5.4 同步报文的生成

根据 CANopen 应用层协议，两个同步报文的发送间隔为一个同步周期。所有的同步 PDO 报文按照同步周期发送。并且，CANopen 管理者框架协议规定同步报文由 CANopen 主站发出。因此，CANopen 主站必须在每一个通讯周期开始时发出一个同步报文，以保证整个网络的正常 PDO 通讯。

根据图 5-3，同步生成算法分为三个状态：初始化状态（INIT）、同步生成状态（GET_SYNC）和等待同步窗口状态（WAIT_WIN）。状态信息保存在任务的私有任务数组 Argu[0]。对于所有的任务，Argu[0]默认为任务的状态保存空间。

```
char SYNCProducer(TaskObj)
  state = TaskObj.Argu[0];                    // 获取当前状态
  switch(state)
    case INIT:                                // 初始化状态
      cobID = GetODEntry(0x1005,0);           // 获取同步报文COB-ID
      syncPeriod = GetODEntry(0x1006,0);      // 获取同步周期
      syncWin = GetODEntry(0x1007,0);         // 获得同步窗口大小
      if(syncPeriod > 0)                      // 如果同步机制使能
        等待一个syncPeriod时间;
        TaskObj.Argu[0] = GEN_SYNC;           // 下一状态为生成同步报文
      else                                    // 同步机制被禁止
        等待无限时间;                          // 使该任务睡眠
      end;
      return TASK_WAIT;

    case GEN_SYNC:                            // 同步生成状态
      发送一个同步报文;
      Global.SYNCWinFlag = 1;                 // 同步窗口开启
      TriggerPDO();                           // 更新所有同步PDO的同步计数
      TriggerMsgSender();                     // 唤醒报文发送任务
      if (syncWin<syncPeriod) && (syncWin>0)  // 如果同步窗口使能
        等待一个syncWin时间;
        TaskObj.Argu[0] = WAIT_WIN;           // 下一状态为等待同步窗口
      else                                    // 同步窗口功能被禁止
        等待一个syncPeriod时间;
        TaskObj.Argu[0] = GEN_SYNC;           // 下一状态为同步生成
      end;
      return TASK_WAIT;

    case WAIT_WIN:                            // 等待同步窗口状态
      Global.SYNCWinFlag = 0;                 // 设置同步窗口关闭
      等待一个(syncPeriod – syncWin)时间;
      TaskObj.Argu[0] = GEN_SYNC;             // 下一状态为同步生成
      return TASK_WAIT;
  end;
```

图 5-3 同步生成任务算法

Fig. 5-3 Algorithm of the SYNCProducer Task

在初始化状态，同步生成任务从对象字典中读取所有的必要信息。根据 CANopen 管理者框架协议，CANopen 允许用户在运行时更改同步周期，使能或禁止同步报文的发送。所有的这些配置都是通过修改对象字典 1005h~1007h 三

个索引项完成。所以，同步生成任务不仅仅为简单的周期执行，还需要实时响应对象字典的修改，为事件驱动方式。

对象字典项 1005h~1007h 的修改将直接触发同步生成任务的执行。这种触发方式需借助于对象字典项（图 3-1）的 pFun 成员变量。该变量为一个函数指针，指向一个特定的对象字典项处理函数。针对于 1005h~1007h 这 3 个对象字典项来说，pFun 指向的函数将触发同步生成任务，并更改同步生成任务结构体的 Argu[0]变量，重定义任务状态为初始化状态，迫使同步生成任务重新读取对象字典的设置。对应的，对象字典项的写函数会主动判断对象字典项的 pFun 指针，如果指针非空，则执行该指针指向的函数。

在读取了对象字典中的配置参数后，如果需要启动同步机制，任务转为同步生成状态。在该状态，任务依次完成发送同步报文，更新同步 PDO 计数器，触发报文发送任务和设置同步窗口等操作。每一个同步 PDO 根据其通讯设置，相隔固定的同步周期发送 PDO 报文，因此每发送一个同步报文，同步生成任务需主动为每一个同步发送 PDO 更新同步计数器。如果该计数器的值大于等于该 PDO 的设定同步周期数，同步生成任务还需触发该同步 PDO 发送任务。

考虑到同步窗口的存在，在特定情况下同步发送 PDO 可能未能在指定的同步窗口内获得总线，从而等待下一个同步窗口。为了优先发送上一个同步窗口内未发送的同步 PDO 报文，同步生成任务会在每一个同步周期主动触发报文发送任务，发送报文以清空同步 PDO 报文发送队列。

如果启动同步窗口机制，在同步生成状态之后，同步生成任务进入等待同步窗口状态。在该状态，同步生成任务将关闭同步窗口，等待下一个同步周期的开始。

5.5 紧急报文处理

紧急报文为优先级最高的报文，但是 CANopen 应用层协议本身并没有定义对紧急报文的具体处理办法，而将紧急报文的处理留给了上层应用软件。所以，CANopen 主站只需要将紧急报文上报给应用软件，由应用软件解析并处理。

基于以上考虑，紧急报文由报文接收任务处理。报文接收任务收到紧急报文后，报文接收任务会直接将报文用消息的方式传递给应用软件。

5.6 PDO 报文处理

PDO 报文的处理分为 3 种情况，PDO 报文的接收，同步发送 PDO 报文的处理和异步发送 PDO 报文的处理。

5.6.1 PDO 报文的接收

PDO 报文为 CANopen 节点间传递过程数据的主要载体。接收 PDO 报文的处理也就是根据该 PDO 的通讯参数和映射参数解析报文，将报文中的数据保存至对象字典的对应位置。从这一点来说，对于 PDO 报文的接收方，同步和异步 PDO 报文并无区别。

从接收 PDO 的处理来看，接收 PDO 报文为简单的顺序执行操作，不需要等待任何事件，因此也无需为每一个接收到的 PDO 报文单独建立任务，完全可以由 PDO 报文分发任务处理。所以，当接收 PDO 报文队列非空时，PDO 报文分发任务被接收报文任务触发，逐一取出队列中的报文，读取该 PDO 的通讯参数和映射参数，解析报文并更新对象字典。

5.6.2 同步 PDO 报文的发送

CANopen 协议栈为每一个发送 PDO 配备一个专门的任务处理，该任务由 PDO 配置函数在新建发送 PDO 时创建。

根据 5.4 节的描述，同步生成任务每发送一个同步报文，会主动更新每一个同步 PDO 发送任务的同步计数器。按照 CANopen 应用层协议的规定，PDO 的发送类型决定了该 PDO 的发送周期[16]。表 5-2 详细列出了 PDO 发送类型的定义。

表 5-2 PDO 的发送类型定义

Table 5-2 Definition of PDO Transmission Type

发送类型	描述
0	同步 PDO，但是同步的方式由具体的设备协议定义
1~240	同步 PDO，每隔传输类型个数的同步周期之后，发送一个 PDO 报文
241~251	不使用
252	PDO 同步发送，但是仅当收到一个远程发送请求之后发送
253	收到远程发送请求之后发送 PDO 报文
254	制造商定义方式
255	异步 PDO，当发生事件时发送

在表 5-2 当中，发送类型 0 和 254 由设备协议和制造商定义，由于 CANopen 主站并没有具体的设备定义，所以将 0 号和 252 号等同，将 254 号和 255 号等同。这样，同步类型的 PDO 包括 0~240 和 252 号 PDO。253~255 号为异步 PDO。

当同步 PDO 发送任务的同步计数器值和发送类型相等的时候，PDO 发送任务会由同步生成任务触发执行。PDO 发送任务检查自身的发送类型和触发条件信息，在当前同步窗口内竞争发送相应的 PDO 报文。

5.6.3 异步 PDO 报文的发送

异步 PDO 报文的发送由事件触发。可能事件有两种，接收到远程发送请求和对象字典项数据的改变。

发送类型为 253 的 PDO 当接收到一个远程发送请求时发送异步 PDO 报文。远程请求报文由 CAN 报文的远程控制位标识（图 2-1）。PDO 报文分发任务当收到一个远程请求报文时，会设置相应 PDO 的触发条件，主动检查该 PDO 的发送类型。如果发送类型为 253，则直接触发该 PDO 的发送任务执行。

相比远程发送请求事件，对象字典项数据改变事件的处理更为复杂。PDO 报文和具体对象字典项之间的映射由 PDO 的数据映射配置决定。然而该映射只能从 PDO 号来查找需要装入 PDO 的对象字典项索引号，为单向映射。从对象字典的索引号无法判定该对象字典项被哪一个 PDO 映射，更无法判断 PDO 的传输类型。

实际上，只有被异步发送 PDO 映射到的对象字典项才需要进行这种反向映射，如果在被异步 PDO 映射的对象字典项的数据中保存 PDO 号，则能解决该问题。但为了这个原因而修改对象字典项的结构体定义，增加一个数据域，将产生大量的内存浪费。考查对象字典项的 pFun 成员变量。所有需要保存 PDO 号的对象字典项都为过程数据，处于 6000h 以上的数据区，一般情况下并无特殊操作，所以 pFun 为空。如果一个对象字典项被映射到异步发送 PDO，该对象字典项的 pFun 成员变量保存对应的 PDO 号，只需要增加一种对象字典项类型即可。这便解释了表 3-2 中的 PDO 映射定义。

根据这种实现方式，对象字典的修改函数在更新对象字典项数据之后，会判断该字典项的类型。如果该字典项被映射到异步 PDO，则根据保存在 pFun 当中的 PDO 号找到对应的发送 PDO 任务，触发该任务执行以发送对应的 PDO 报文。

5.7 SDO 报文处理

SDO 通讯用于主站对从站对象字典的配置过程，比如启动从节点的节点维护机制，配置从节点的 PDO 通讯参数等等。本文的 CANopen 主站不支持程序下载，因此也不需要通过 SDO 传递大量的数据。所以，只实现了最简单的 SDO 加速传输方式。

在这种传输中，CANopen 主站总是发起 SDO 通讯。从站响应主站发出的 SDO 请求，执行相应操作，返回一个 SDO 报文。在这种方式下，CANopen 主站没有必要为每一个存在的 SDO 通讯建立一直存在的 SDO 处理任务，只需要为当前正在通讯的 SDO 建立任务。

整个 SDO 通讯的实现过程可由图 5-4 表示。SDO 通讯的发起者新建 SDO 处理任务。SDO 处理任务根据发起者的要求生成 SDO 报文放入 SDO 报文发送队列，并触发报文发送任务执行。报文发送任务将 SDO 报文发送到 CAN 总线。正常情况下相应的 CANopen 节点会收到该 SDO 请求并返回一个 SDO 响应报文。该响应报文被报文接收任务送入接收 SDO 报文队列，导致 SDO 报文分发任务被触发。SDO 报文分发任务根据 SDO 号触发对应的 SDO 处理任务。SDO 处理任务解析该响应报文，将 SDO 请求的执行情况和结果传递给 SDO 通讯发起者，通讯完成。如果出现异常，比如 SDO 通讯超时，SDO 处理任务会在 SDO 超时后被重新调度运行。SDO 处理任务将超时的结果传递给 SDO 通讯发起者。

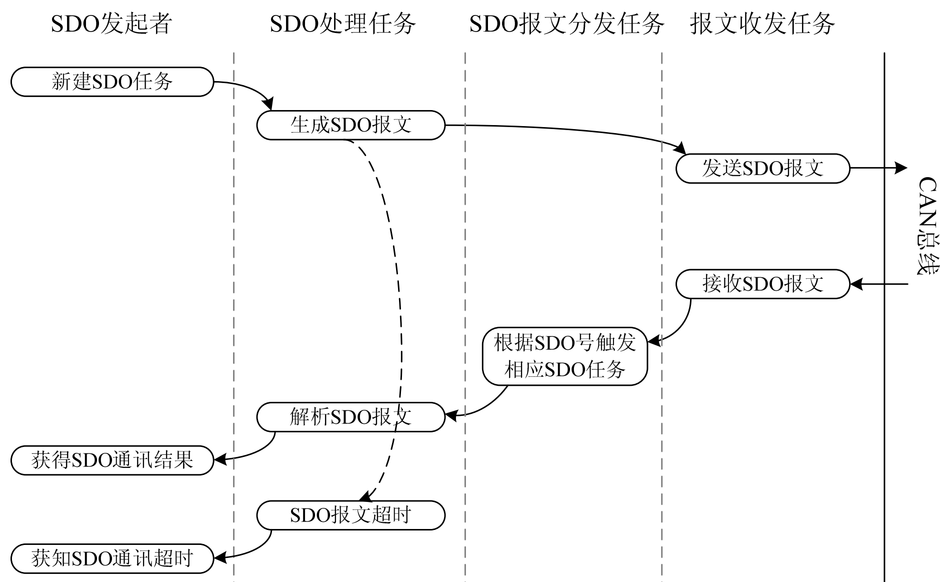


图 5-4 SDO 通讯过程

Fig. 5-4 Process of SDO Communication

5.8 节点状态的维护

CANopen 规定了两种节点状态维护机制，心跳报文机制和节点保护机制。每一个从节点必须实现其中一种，CANopen 网络允许网络中的节点采用不同的状态维护机制。

5.8.1 心跳报文机制

心跳报文机制是当前 CANopen 协议推荐使用的节点状态维护机制。使用心跳报文机制的节点，间隔一个固定周期，将主动向 CANopen 网络发出一个节点状态报文。需要监视该节点运行状态的节点可接收此节点状态报文来监视该节点。如果超过一个周期没有收到状态报文，说明该节点意外脱离 CANopen 网络。

CANopen 主站为 CANopen 网络的心跳报文接收者之一，通过心跳报文监控网络中的节点。对于每一个进入 CANopen 网络的从节点，如果采用心跳报文机制，CANopen 主站会在启动检查的过程中启动该从节点的心跳报文机制，并建立心跳报文监视任务。该任务的等待超时时间为被监视节点的心跳周期。正常情况下，在任务等待超时之前，CANopen 主站会收到该节点的心跳报文。监视任务解析该报文，更新对象字典 1F82h 位置的对应子索引，并重置超时时间。如果超过一个心跳周期 CANopen 主站都没有收到心跳报文，该监视任务会被调度机触发，产生一个超时错误通知上层应用程序。

5.8.2 节点保护机制

节点保护机制是 CANopen 协议早期定义的节点状态维护机制，被一些早期的 CANopen 节点使用。一般情况下，只有当从节点不支持心跳报文机制时，CANopen 主站才会使用从节点的节点保护机制。

采用节点保护机制的从节点被动地响应 CANopen 主站的状态请求。间隔固定时间，CANopen 主站向从节点发出节点状态请求的远程请求报文。从节点收到该请求后返回节点状态报文。CANopen 主站通过该状态报文监视从节点。从另一个角度，如果从节点超过该固定时间间隔仍未收到主站的状态请求，说明从节点自身意外脱离网络或主站出现异常。从节点也可通过该机制监视 CANopen 主站的状态。

和心跳机制一样，每一个进入 CANopen 网络的从节点，如果采用节点保护

机制，CANopen 主站会在启动检查的过程中启动该从节点的节点保护机制，并建立节点保护监视任务。该任务周期性地执行，周期性地发送从节点状态请求远程请求报文。如果一段时间后没有收到从节点的状态回复，CANopen 主站协议栈将上报应用程序该节点的异常，否则解析从节点的状态报文，更新对象字典 1F82h 位置的对应子索引。

5.9 网络启动过程

如图 5-5，CANopen 网络的启动过程分为 3 个阶段，CANopen 主站启动、通讯配置和从节点启动。CANopen 主站启动过程将初始化 CANopen 主站。之后的通讯配置阶段，CANopen 主站将根据用户的配置信息和从网络自动获得的信息配置与已知节点的 PDO 通讯。然后，CANopen 主站将利用从节点启动过程检查从节点的通讯参数信息，启动从节点的节点状态维护机制。最终，所有的从节点转变为运行状态，CANopen 网络进入正常运行状态。

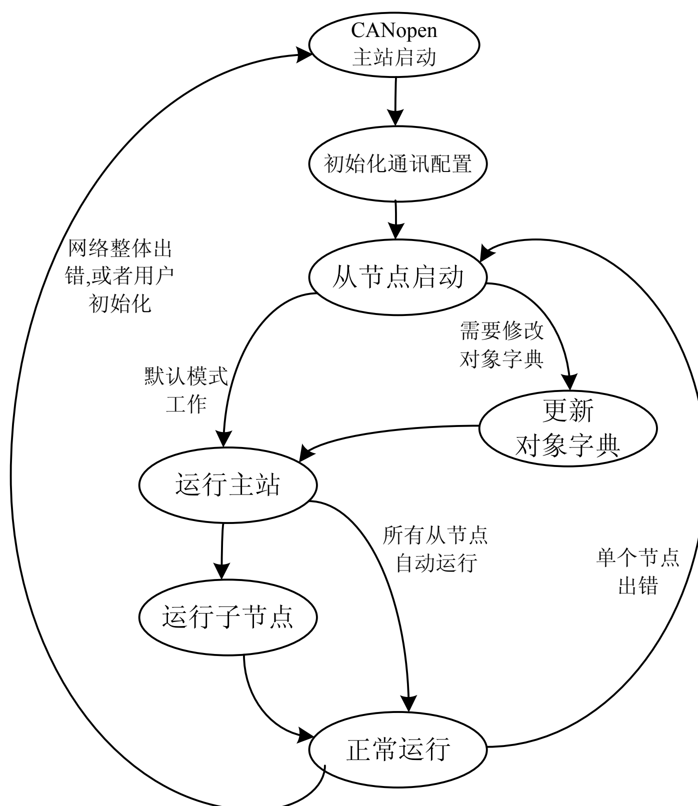


图 5-5 CANopen 网络的运行状态转换图

Fig. 5-5 Status Chart of CANopen Network

5.9.1 CANopen 主站的启动

CANopen 主站上电之后，CANopen 主站的启动过程将依次完成初始化全局数据对象结构、初始化时钟、初始化对象字典空间、初始化 CAN 控制器和启动常置任务的操作。其中全局数据对象结构保存了所有的全局数据、任务结构体、节点状态结构体等关键全局数据对象。在启动阶段被初始化的常置任务包括报文接收任务、报文发送任务、SDO 报文分发任务、PDO 报文分发任务、NMT 报文分发任务和同步生成任务。

5.9.2 通讯配置

CANopen 主站启动完成之后，CANopen 调度机和对象字典就已经正常工作。为了启动网络，CANopen 主站应首先获得 CANopen 网络当中的节点信息和通讯方式配置信息。CANopen 网络的通讯方式配置有两种方式：用户配置和主站自动识别。用户配置即上层应用软件通过相应软件接口主动将已知节点的信息添加入 CANopen 主站的对象字典，通过 CANopen 主站配置从站的对象字典建立 PDO 通讯。自动识别的方式则是 CANopen 主站通过监听网络主动发现节点，并按照 CANopen 的默认方式配置通讯。按照 CANopen 标准，从节点上电时会向 CANopen 网络发出启动报文。CANopen 主站接收到启动报文后，就可通过启动报文的节点号获知新节点进入网络，并用默认的通讯参数初始化对象字典空间。

在通讯配置中，PDO 的配置为 CANopen 主节点和从节点之间的交互过程。为了保证 CANopen 网络的兼容性，PDO 配置必须尽可能地支持不同节点的配置方式。整个配置的交互过程如图 5-6。

通过读取从站 PDO 的 COB-ID，CANopen 主站可以判断出该 PDO 是否存在。如果 COB-ID 读取正确，说明目标 PDO 存在。根据 CANopen 基本通讯协议，CANopen 主站首先关闭该 PDO，即改写存储 COB-ID 的对象字典项的第 31 比特为 1。然后，CANopen 主站根据配置要求改写传输类型，并最终改写 COB-ID，重新使能 PDO，整个 PDO 配置工作完成。但是在测试中发现，某些厂家的 PDO 并不支持 PDO 关闭操作。为了兼容这些特殊要求，当 CANopen 主站发现 PDO 不能关闭时，只要传输类型可以改为目标值或者已经等于目标值，同时 COB-ID 正确，CANopen 主站也能正常完成 PDO 的配置工作。

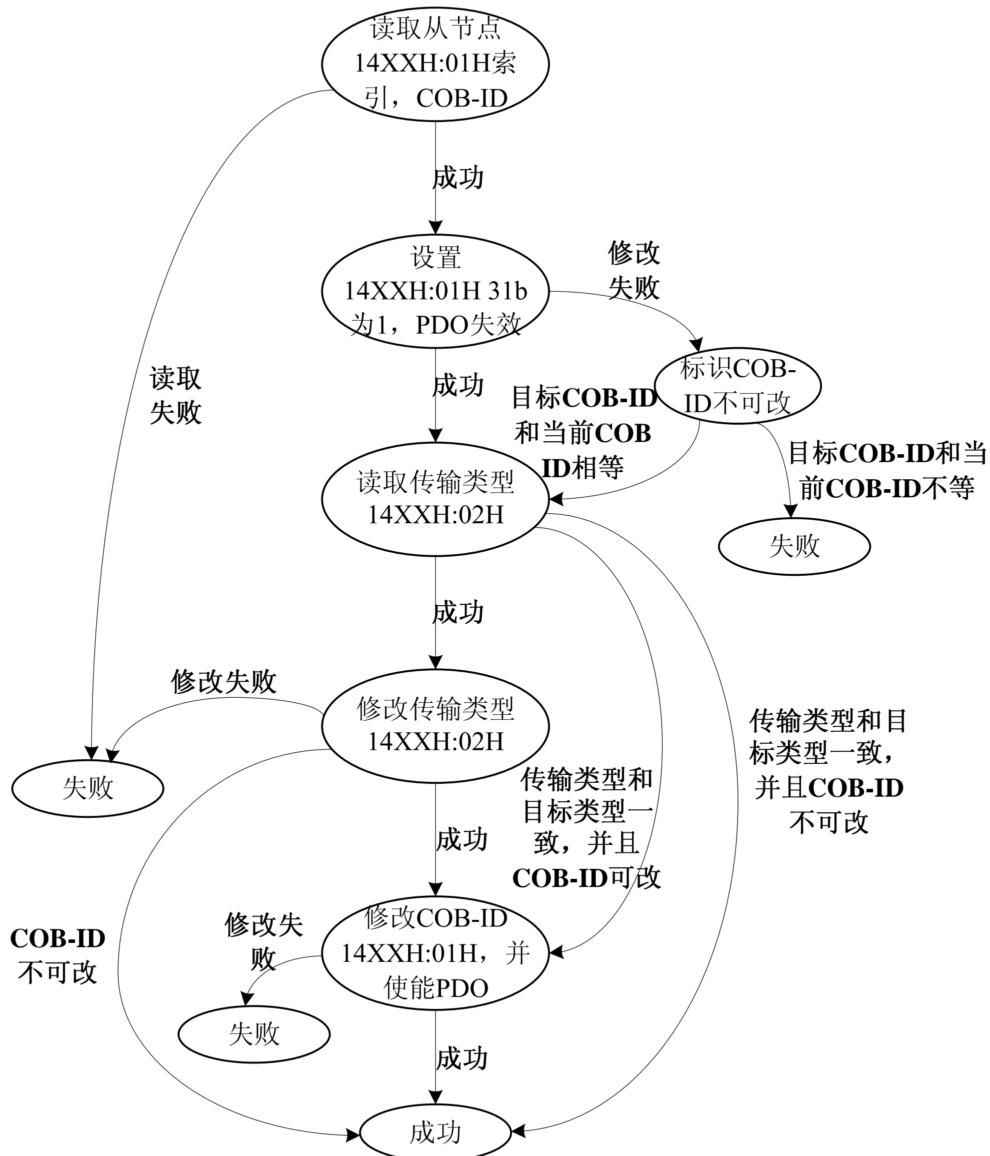


图 5-6 PDO 的配置过程

Fig. 5-6 Config Process of PDO Communication

5.9.3 从节点启动过程

通讯配置完成后，CANopen 主站已经获得所有进入 CANopen 网络的从节点节点号，并完成了 PDO 通讯配置。按照 CANopen 管理者框架协议的规定，CANopen 主站在启动从节点前，需要逐一验证从节点的配置信息，启动节点状态监视机制。根据该协议，CANopen 主站当接到上层应用程序的启动网络命令后，需要为每一个已记录的从节点启动一个节点启动任务。图 5-7 简要地给出了该节点启动任务的执行流程。

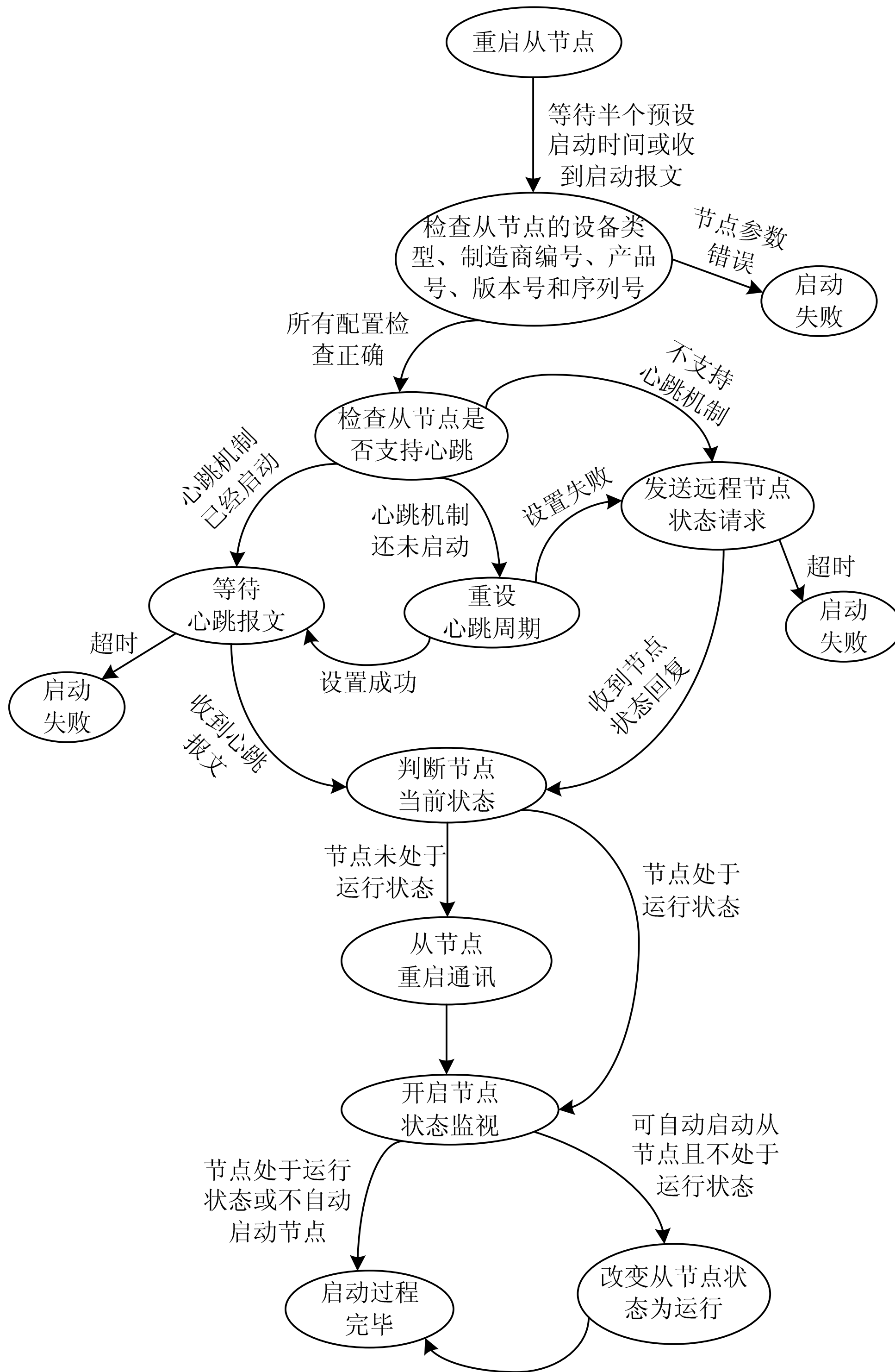


图 5-7 节点启动任务流程

Fig. 5-7 Process Flow of Node Start Task

所有从节点的节点启动任务完成后，CANopen主站主动收集所有任务的执行结果。正常情况下，所有节点启动正确，CANopen主站对整个CANopen网

络发出启动命令，网络进入正常工作状态。如果出现节点启动失败，CANopen将根据对象字典中的配置信息和节点启动失败的类型，上传应用程序相应错误信息，并有选择地启动网络，甚至将失败的节点剔出网络。

5.10 本章小结

由于调度机的使用，CANopen 主站协议栈大量地运用任务来完成各个功能。不同模块的任务具有不同的优先级，保证了高优先级的功能优先完成。同时，各个任务之间采用触发的方式进行同步，保证了整个协议栈的协同工作。

本章在解释了各个功能模块运作方式的同时，着重解释了同步报文的生成，高兼容能力的 PDO 配置方法和兼容 CANopen 管理者框架协议的从节点启动检查过程。

第六章 CANopen 主站的实现及测试

6.1 简介

本章将在第五章 CANopen 协议栈的基础上，分析如何建立一个 CANopen 主站。首先，本章将分析 CANopen 主站的整体结构，讨论关于实现的具体问题。然后，在实际的测试网络中得到关于 CANopen 的实时性、节点检查和协议完整性等方面的测试结果。最后将给出 CANopen 协议栈的移植方法。

6.2 主站测试平台

6.2.1 基于 WindowsXP 的 CANopen 主站设计

为了测试 CANopen 主站协议栈的功能和性能，我们将该协议栈运用在 Windows 系统上，实现了一个具备基本功能的原型测试系统。整个测试系统的软件架构如图 6-1 所示。

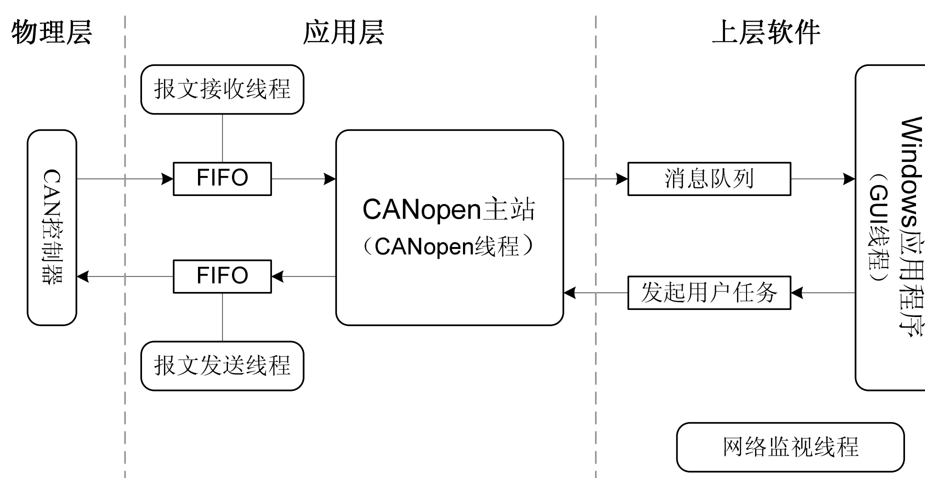


图 6-1 CANopen 主站测试系统的软件架构

Fig. 6-1 Software Architecture of CANopen Master Station Test Platform

如图 6-1 所示，整个测试系统分为 3 个层次：由 CANopen 网络和 PCI-CAN 接口卡[52]组成的物理层、由报文接收发送线程与 CANopen 主线程组成的应用层、Windows GUI 所代表的上层应用软件。

CANopen 主站的物理层由 PCI-CAN 转接卡和相应的 PCI 驱动函数构成。该 PCI-CAN 转接卡一共提供两个 CAN 接口，CANopen 主站使用其中一个接口，另外一个接口直接连到上层应用软件实现对 CANopen 网络的报文监控。两个 CAN 口都直接连接到实际的测试网络。整个测试网络包含 4 个 CANopen 节点，如表 6-1 所示。

表 6-1 CANopen 测试环境节点信息

Table 6-1 Node Information of CANopen Test Network

协议栈名称	节点类型	硬件平台	协议栈
CANopen 主站	主站	Pentium-4 3.0G，WindowsXP	自主开发
BK5120 总线耦合器	从节点	Beckhoff 成品节点	未知
汽车仪表节点[12]	从节点	ARM9 Linux	Festival[25]
电动机节点[13]	从节点	DSP，TMS320F2812，无操作系统	MicroChip CANopen[26]

应用层由第五章所述的 CANopen 主站协议栈实现。由于 WindowsXP 操作系统有自己的线程调度机，CANopen 主站协议栈只能在一个较高优先级的线程中运行。同样因为 WindowsXP 操作系统的存在，中断不能被用户模式的软件控制，因此在图 5-1 中原本应当由中断服务程序实现的 CAN 驱动部分由两个高优先级的 CANopen 报文接收线程和 CANopen 报文发送线程线实现。

上层软件在测试系统中提供了一个网络配置和动态控制的图形用户接口（GUI）。如图 6-2，通过上层软件的 GUI，用户可以查看主站和从站的所有对象字典项，手动向网络中添加一个节点，手动配置从节点的 PDO 通讯参数，和改变整个网络或者其中一个从节点的运行状态。

另外，通过 PCI-CAN 转接卡提供的第二个 CAN 接口，上层软件用一个单独的线程实现了网络监视窗口。通过该监视窗口，用户可以分析网络上的真实 CAN 报文通讯情况，协助调试和数据分析。

6.2.2 线程间的数据通讯

在 WindowsXP 操作系统下运行的 CANopen 调度机实际上是操作系统的一个线程。WindowsXP 是一个抢占式的操作系统，调度机可能在某些时刻被操作系统打断。这样就带来了两个问题：线程间的共享数据保护和调度机的实时性。

根据各线程的实时性要求，表 6-2 给出了各线程的优先级分配。

CAN 报文的收发线程显然会打断调度机的执行。根据 5.3 节的叙述，调度机与报文收发线程之间使用 FIFO 作为通讯媒介，同时调度机对所有任务队列的相关操作加锁，因而 CAN 报文的收发线程并不会破坏共享数据。

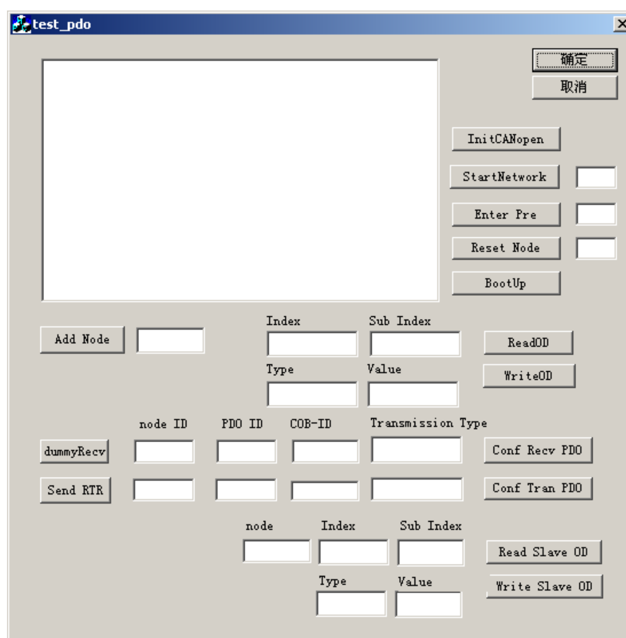


图 6-2 测试系统用户界面

Fig. 6-2 Graphic User Interface of Test System

表 6-2 CANopen 测试系统的线程优先级

Table 6-2 Priorities of the threads in CANopen Test Platform

线程	优先级	与其通讯的线程
报文接收线程	HIGHEST	CANopen 主线程
报文发送线程	HIGHEST	CANopen 主线程
CANopen 主线程	ABOVE_NORMAL	报文接收线程、报文发送线程、用户界面
用户界面	NORMAL	CANopen 主线程
监视线程	NORMAL	无

尽管用户界面线程的优先级比调度机线程低，但是由于 WindowsXP 采用动态可变优先级机制，导致用户界面线程还是可能打断调度机线程。此外，用户界面和调度机之间的共享数据较多，包括任务空间、对象字典和 SDO 通讯的数据区。显然，对所有数据进行保护是不切实际的。考虑到调度机已经对任务队列进行保护，这里采用用户界面生成 CANopen 任务代替其完成操作的方法。由于用户界面和调度机之间的操作由 CANopen 任务完成，因此用户界面和中断服

务程序一样，只需要修改调度机的任务队列。用户界面生成的任务完成操作后，任务会通过消息队列将执行结果返回给用户界面。消息队列采用 FIFO 的方式，因此也不会出现数据冲突问题。

关于调度机的实时性，在有操作系统的情况下，调度机可能会被其它低优先级任务打断。不过，并非任何时刻打断调度机都会影响 CANopen 的实时性，只有当 CANopen 确实在处理任务的时候，被操作系统打断会导致事件的响应时间加长。如果调度机采取在无操作时主动睡眠的方式，主动选择被打断的时机，将能够减少在执行任务时被打断的概率，从而维持相对的实时性。为了实现该操作，我们必须修改 4.3.2 小节所述的调度机算法和 *MoveTaskToRun()*方法，其中调度机算法如图 6-3 所示。

```
void scheduler(RunQueue RQ, TaskQueueHeader WQ)
{
   while(1) {
      sleepTime = MoveTaskToRun(RQ,WQ);        // 更新可执行任务
      if(SelectTaskForProc(&pTRun, RQ)) {        // 选择任务
         Sleep(sleepTime );                    // 无任务可执行
         continue;
      }
      rV = (pTRun->pFun)(pTRun);               // 执行任务
      switch(rV) {
         case TASK_OK:                         // 执行完毕，释放空间
            释放任务空间; break;
         case TASK_RUN:                        // 主动放弃执行，重新调度
            将任务放入运行任务队列; break;
         case TASK_WAIT:                       // 等待条件，返回等待
            将任务放入等待任务队列; break;
         default:                              // 异常退出
            return;
      }
   }
}
```

图 6-3 改进的调度机算法

Fig. 6-3 Modified Scheduling Algorithm

在改进的算法中，*MoveTaskToRun()*方法将返回一个可睡眠时间。如果在当前调度周期没有新的可执行任务，那么 *MoveTaskToRun()*方法返回等待任务队列中头节点的等待时间，否则，返回 0。进而，如果 *SelectTaskForProc()*方法返回假，说明运行队列当前没有任务，那么，*MoveTaskToRun()*方法返回的时间就是在没有新事件发生的情况下，调度机的最大可空闲时间。所以，可以在这个时候放弃运行时间片[53]，迫使操作系统的调度机重新计算时间片，让低优先级的

线程执行，并且对实时性无任何伤害。这样，低优先级的线程则获得了可执行的时间片，WindowsXP 的调度机不需要再调整用户界面的线程优先级，自然减少了低优先级打断调度机运行的概率。

该方法不仅可用于 WindowsXP 操作系统，实际上可运用于任何支持动态变优先级的操作系统中。

6.3 测试数据分析

6.3.1 实时性

CANopen 网络正常工作之后，所有的实时数据通过 PDO 通讯传输。从硬件的角度，同步 PDO 较容易实现；从控制系统的角度，利用同步 PDO 更新实时性较高的数据也便于管理，并可减少数据的总体传输量。因此实时数据一般选择利用同步 PDO 传送。根据 CANopen 应用层协议，同步 PDO 会依据同步周期发送，PDO 的最高发送频率实际上为同步帧的发送频率。所以，CANopen 主站的实时性能取决于单个同步周期内 CANopen 主站可以处理的 PDO 报文个数。

从实现来说，PDO 报文的解析直接由 PDO 分发任务完成。那么，PDO 报文的处理时间包括 PDO 报文接收任务的执行时间、PDO 分发任务的执行时间、和调度机的调度时间。对于每一个 PDO 来说，PDO 的处理时间基本不变。

通过在代码中添加计时单元，实际测量从发生接收报文中断到 PDO 解析完毕所耗费的时间，可以得到 CANopen 主站在不同同步周期下，PDO 的极限处理速度，如图 6-4。

根据参考文献[31]中的数据，混合动力汽车控制系统的数据更新速率为 1k~2kHz。在 CANopen 协议中，实时数据通过 PDO 传输，数据更新速率为 1k~2kHz，即要求当同步周期小于 1ms 时，CANopen 主站能够处理至少一个 PDO 报文。根据图 6-4，即使在较慢的 ARM7 平台上，CANopen 主站数据更新率也能达到 5kHz，大大超出了混合动力汽车控制系统的实时要求。

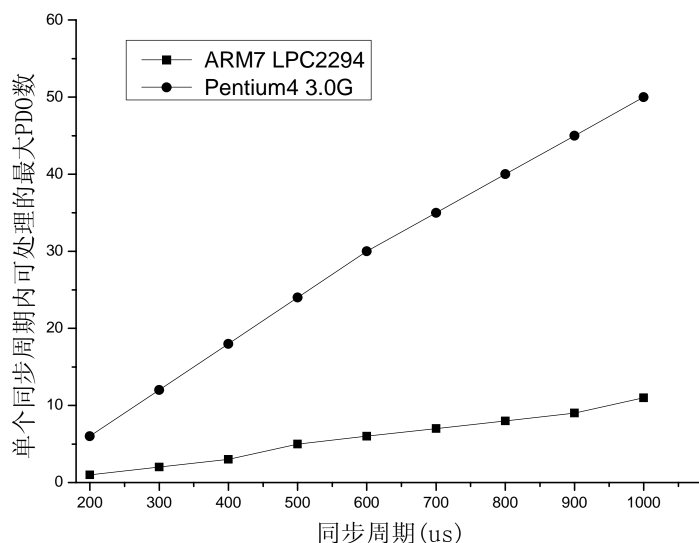


图 6-4 单同步周期可处理 PDO 数量

Fig. 6-4 the Number of PDOs Processed in One Synchronization Period

还应指出，一个 CANopen 报文的最短长度为 52 比特。即使在最高的 1Mbps 总线传输速率下，传输一个最短的 CANopen 报文也需要 52μs。那么，在 200μs 的同步周期中，物理层最多能够传输 4 个 CAN 报文。在 WindowsXP 平台上 CANopen 的处理速度已经超过了物理层的速度极限。

6.3.2 WindowsXP 平台上 CANopen 主站的速度

连接标准的 Beckhoff 公司的 BK5120 总线耦合器，表 6-3 给出了在 CANopen 主站和 BK5120 总线耦合器每同步周期各发出一个 PDO 报文，BK5120 总线耦合器每隔 1s 发送一个心跳报文的情况下，不同的同步通讯周期对系统的影响。所有的数据通过在 WindowsXP 系统下，使用 500Kb/s 的 CAN 总线通讯速率实际测得。

在表 6-3 中，I/O 通讯量为操作系统访问外设的通讯量，基本正比于 CAN 总线上的带宽使用量。从表可见，当同步周期大于 1ms 时，CANopen 主站的 CPU 占用率极低，CANopen 主站的运行基本上对操作系统没有任何影响。当同步周期小于 1ms，CANopen 主站的 CPU 占用率上升。通过软件检查，此时 CPU 主要被 CANopen 主站中的报文发送线程所占用。其原因显然是由于 CAN 总线上通讯繁忙，导致报文发送线程经常性的发送失败，需要重新发送。当同步周期等于 300μs 时，由 BK5120 总线耦合器发出的低优先级的节点状态报文出现抢占总线困难，导致主站开始检测到从节点状态超时。当同步周期等于 200μs

时，CANopen 主站已经到达了处理极限。大量的发送 PDO 被同步报文抢占而不能发送。CANopen 线程内的 PDO 发送报文队列长度不断加长，导致系统内存占用增长。从主站的 GUI 界面上可以看出，虽然数据通讯仍在继续，但是 PDO 通讯显然已经出现困难。不过，在 BK5120 总线耦合器的最大支持速率（500Kb/s）下，200μs 内 CAN 总线上只能发送 100 比特数据，显然是不可能发送包括同步帧在内的 3 个 CAN 报文，因此系统的不稳定源于物理连接的限制而并非主站的性能。

表 6-3 同步周期对系统的影响

Table 6-3 System Response to Different Communication Periods

同步周期	I/O 通讯量	CPU 占用率	系统异常
500ms	8.3KByte/s	0.0%	无
250ms	16.5KByte/s	0.0%	无
100ms	37.1KByte/s	0.0%	无
50ms	66.0KByte/s	0.0%	无
10ms	264.1KByte/s	0.0%	无
1ms	2.7MByte/s	0.0%	无
500μs	5.6MByte/s	100%	无
400μs	7.2MByte/s	100%	无
300μs	9.4MByte/s	100%	出现节点状态超时
200μs	10.8MByte/s	100%	内存占用不断增大

6.3.3 WindowsXP 平台上 CANopen 主站的内存占用

连接标准的 Beckhoff 公司的 BK5120 总线耦合器，图 6-5 给出了在不同 PDO 数量时，CANopen 主站的内存占用情况。

当 PDO 数量为 0 时，CANopen 主站的内存使用即为初始内存大小。应当说明，在 WindowsXP 系统下的内存使用和直接在嵌入式系统下的内存占用是不一样的。WindowsXP 的应用程序往往附带了库函数的执行代码，并且这部分内存也包括了 GUI 界面的内存，因而要远大于嵌入式系统下的内存占用。

初始情况下，CANopen 主站的内存占用约为 1.1MByte，相比其它的软件，和商业的 CANopen 主站，显然是很小的。随着 PDO 数量的增加，内存占用也不断增加。增加的原因是 CANopen 主站需要为每一个 PDO 对象在对象字典中开辟相应的数据区间，并在等待任务对列中添加相应的 PDO 处理任务。需要指

出，这里所说的 PDO 为发送 PDO，对于接收 PDO，CANopen 主站不需要建立相应的处理任务，因而内存占用会更小。图 6-5 中的曲线不连续问题是由于操作系统的动态内存分配机制。少量的动态内存分配有时并不需要操作系统分配一块新的虚拟内存。

分析图 6-5，我们可以得出如下结论：在 WindowsXP 平台上的 CANopen 测试主站，每分配一个 PDO，需要数据空间约 800Byte。需要指出，这里的 800Byte 并不代表在嵌入式系统里也需要 800Byte。WindowsXP 使用虚拟内存管理，这里的空间为虚拟空间，实际物理空间应小于该值。

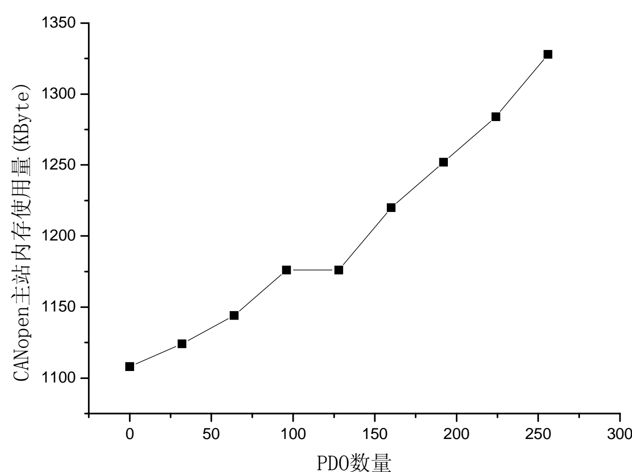


图 6-5 PDO 的内存占用

Fig. 6-5 Memory Occupation of PDOs

6.3.4 节点启动检查

基于测试系统，CANopen 主站对 Beckhoff 公司的 BK5120 总线耦合器的节点启动检查通讯过程如图 6-6。

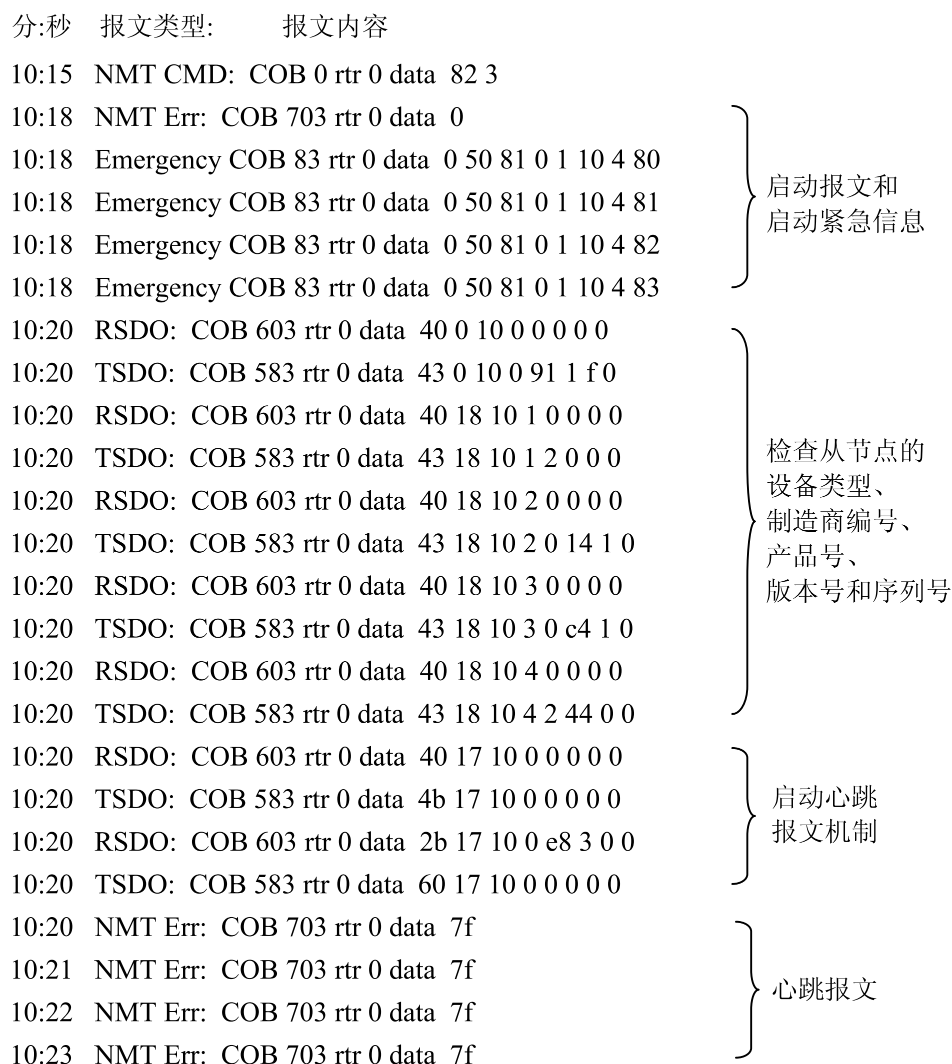


图 6-6 实际的节点启动报文序列

Fig. 6-6 Field Message Sequence of Node Boot Procedure

通讯过程中，CANopen 主站在 10:15 时发出了节点重启命令。3 秒后，主站在 10:18 时收到了从节点的启动报文。然后，CANopen 主站在 10:20 时的 1 秒钟内依次查询了从节点的设备类型(1000h)、制造商编号(1018.01h)、产品号(1018.02h)、版本号(1018.03h)和序列号(1018.04h)。检查完成后，CANopen 主站通过修改从节点的 1017h 位置启动了心跳报文机制。从 10:20 时到 10:23 时间，CANopen 主站收到了间隔 1 秒的心跳报文，并显示从节点当前处于预运行状态。

整个通讯过程说明 CANopen 主站能够正确地检查从节点信息和启动节点的节点状态维护机制，最终完成 CANopen 管理者框架协议规定的节点启动过程。

6.3.5 协议支持的完整性

本文所述的 CANopen 主站协议栈实现了 CANopen 应用层协议规范[7]和 CANopen 管理者框架协议[22]中网络管理主站的功能。相比开源社区的 CanFestival CANopen 主站[25]，本文的 CANopen 主站具有更多的功能和更高的配置灵活性。此外，CANopen 主站的可移植性也是其它商业 CANopen 主站所不能比拟的。表 6-4 具体给出了两个 CANopen 主站对 CANopen 协议的支持比较。

CanFestival 项目设计的 CANopen 协议栈是当前开源协议栈中功能最全的主从站通用版本。即使这样，从表 6-4 也可看出，CanFestival 仅支持 CANopen 应用层规范中规定的部分内容，其主站的实现部分也没有参照 CANopen 管理者框架协议。相比之下，本文提出的 CANopen 主站对 CANopen 应用层规范的支持更加完整，同时按照 CANopen 管理者框架协议实现了网络管理者主站。另外，灵活的任务管理机制提供了对通讯对象、数据字典等核心内容的动态配置和管理功能，这是基于任务循环结构的开源协议栈所不能比拟的。

表 6-4 对协议的支持程度比较

Table 6-4 Comparison of the Protocol Compatibility

	功能	必要性	CanFestival	本文的 CANopen
CANopen 应用层和通讯协议	SYNC 的生成	必要	完成	完成，最小 200 微秒
	SDO 客户端	必要	部分，仅支持加速方式	部分，仅支持加速方式
	SDO 服务器端	必要	完成，仅支持 1 个	完成，可支持多个
	PDO 发送	必要	完成	完成
	PDO 接收	必要	完成	完成
	心跳报文机制	必要	完成	完成
	节点保护机制	必要	部分完成	完成
	从节点状态控制	必要	完成	完成
	紧急报文	必要	不支持	通知上层
	时间戳	可选	不支持	不支持
CANopen 管理者框架协议	节点启动检查	可选	不支持	部分，除配置检查的所有部分
	配置及软件下载	可选	不支持	不支持
	第三方 NMT 命令	可选	不支持	完成
	SDO 通道建立	可选	不支持	不支持
	主站竞争机制	可选	不支持	不支持
其它	PDO 和 SDO 最大支持数量		宏配置，运行时不可改	支持 CANopen 规定的最大数量，受系统运行速度制约
	对象字典的类型		支持 1，2，4 字节类型和字符类型	支持 1，2，4 字节类型
	对象字典的修改		完成	完成
	对象字典项的新建与删除		不支持	完成
	直接对从节点对象字典操作		不支持	完成
	可移植性		仅支持类 linux 操作系统	部分修改后可运行于所有支持 C 编译器的操作系统和无操作系统平台

6.4 关于可移植性的考虑

本文所述 CANopen 主站基于调度机实现任务的并行处理。调度机由标准 C 编写，没有借助任何汇编或者特定操作系统的系统调用，任何标准 C 编译器都可编译。但是，在不同平台上，还是有很大一部分的代码需要重写，这也是不

基于操作系统的可移植软件不可避免的问题。缓解办法是严格地规划文件和代码结构，使得移植的工作负担减到最小。

整个 CANopen 主站协议栈的代码结构如下（图 6-6）：

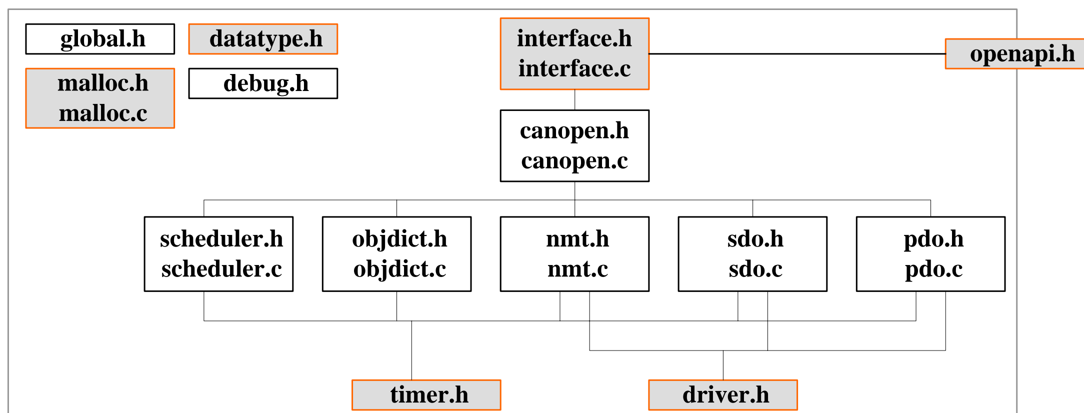


图 6-6 CANopen 主站协议栈的代码结构

Fig. 6-6 Source File Structure of CANopen Master Stack

整个协议栈的核心部分由图 6-6 中间排成一排的 5 个模块组成。它们分别实现了调度机、对象字典、网络管理、服务数据对象和过程数据对象的相关功能。这些模块依靠微秒级时钟（timer）和 CAN 控制器的驱动函数（driver）工作。不同平台和不同的 CAN 控制器的接口显然无法统一，所以这里定义了统一的接口函数。只要能在不同平台上提供符合定义的时钟和控制接口函数，上面的 5 个核心模块就能正常工作。

在不同的 C 编译器下，数据类型的定义也不一样。有的编译器定义整型（int）为 16 比特，而大多数的则定义为 32 比特。为了排除编译器对协议栈数据字长的影响，datatype.h 定义了独立于平台的标准数据类型，其它各文件只使用 datatype.h 文件定义的标准数据类型。在不同编译器下，只需要更改 datatype.h 文件，即可完成数据类型的本地化。

CANopen 协议栈的灵活性建立在动态内存分配之上，然而动态内存分配并非标准 C 函数，不同系统和编译器提供的动态内存分配函数功能也不尽相同。为了能在不提供动态内存分配支持的编译器上编译协议栈，编写了一个简易的不依赖于第三方库的 malloc 模块。该模块可支持最大 1M 内存的动态管理（附录 4）。

此外，global.h 定义了全局数据对象，debug.h 定义了底层的调试宏，都是标准 C 文件，不存在移植问题。

在 5 个核心模块之上是 canopen 模块。该模块可以看成 CANopen 协议栈的主函数。它负责调用调度机模块的执行函数，维护调度机的运行。并且，它使用由接口模块（interface）提供的加锁和解锁函数，对任务队列进行保护。

Interface 模块在 canopen 之上，如果没有操作系统，它提供简单的加锁解锁机制和与上位机的通讯接口。如果有操作系统，它则提供所有的锁机制、消息机制和系统队列。当然，CANopen 协议栈需要向上层应用软件提供标准的应用程序接口。根据 6.2.2 小节的叙述，应用软件通过建立任务的方式执行对 CANopen 的访问和配置。然而建立任务对于应用程序来说过于复杂，协议栈应当提供简单的函数型接口。所以，接口模块为每一种用户需求编写建立任务和配置任务的过程，以简单函数的方式提供应用程序接口。openapi.h 文件则将接口模块中的接口函数打包，用本地数据来重新定义。比如在基于 WindowsXP 的测试系统中，openapi.h 则将所有的接口函数封装成动态链接库，应用程序只需要引用 openapi.h 的一个副本，就可以通过动态链接库运行 CANopen 协议栈。

在严格的文件定义之后，协议栈的移植变为根据具体平台提供时钟、数据定义、CAN 控制器驱动和按需编写应用程序接口等无法脱离平台的工作，大大降低了移植难度。

至论文完成时，该 CANopen 主站协议栈已经被成功地移植到基于 MC9S12DP512 的 μC/OS-II 嵌入式操作系统中[54]。

6.5 本章小结

通过实际的网络测试，本文所述的 CANopen 主站完全达到了混合动力汽车的实时性要求，完全支持 CANopen 管理者框架协议[22]中所规定的节点检查过程，并且在性能与协议支持完整度上远超过了开源协议栈 CanFestival[25]。另外，在协议栈移植方面，严格的文件分割降低了协议栈的移植难度。

结 论

本文通过分析 CAN 总线所具有的严格优先级和高度实时性的运行特点，CANopen 应用层协议具有的高度灵活性和可配置性，以及混合动力汽车整车控制系统高达 2kHz 的数据更新率，得出了 CANopen 网络配置者主站必须同时具备实时运行、并行处理、灵活配置和可移植的能力。

基于以上四个因素以及 CANopen 网络自身的特点，本文提出了两个新设计：基于散列表的对象字典和基于标准 C 语言非抢占式任务调度机的 CANopen 主站协议栈。

基于散列表的对象字典有效地克服了传统数组型对象字典可配置性差的问题。借鉴 Cache 的方法，通过访问次数对散列表的溢出表实时排序，大大减少了散列算法在速度上和数组方法的差别，使得散列表方法具有和传统数组方法类似的快速读取特性。

标准 C 语言实现的非抢占式任务调度机为 CANopen 事件的并行处理提供了良好平台。同时标准 C 语言的内部实现决定了该方法具有高度的可移植性。基于调度机实现的 CANopen 主站，相比无操作系统的任务循环方法，具有更高的实时性和功能的独立性；相比基于操作系统线程或者进程的实现方法，具有更好的可移植性。

经过实际网络测试，本文提出的 CANopen 主站方法完全达到了混合动力汽车控制系统提出的 1~2kHz 数据更新率的实时要求，同时在协议实现的完整度上大大超出了开源协议栈的水平。可移植性更拓宽了该方法的使用范围。

鉴于汽车控制系统硬件环境和控制要求，本主站没有提供 EDS 文件的支持；SDO 通讯只实现了最根本的加速通讯方式。但是，这些功能和本文提出的 CANopen 主站并没有本质冲突。所以当该主站适用于其它控制系统时，这些功能可以被较容易地添加到主站协议栈中。

参考文献

1 孔峰, 张衡, 宋雪桦, 等. 基于CANopen协议的汽车控制网络初探. 汽车工程. 2007, 29(7): 594-596, 605

2 林长加. CAN总线技术在混合动力汽车中的应用. 大连理工大学硕士学位论文. 2008

3 聂单根. 基于CAN/LIN总线的汽车车身网络技术的应用研究. 浙江大学硕士学位论文. 2007

4 U. Kiencke, S. Dais and M. Litschel. Automotive serial controller area network. *Transactions of the Society of Automotive Engineers*. 1986, 95(2): 823-828

5 ISO. *ISO11898-Part1. Road vehicles - interchange of digital in- formation - part 1: Controller area network (can) for high- speed communication*. 1993

6 苏剑, 罗峰, 袁大宏. 控制器局部网络在汽车中的应用. 汽车技术. 2003, (5): 1-4, 25

7 CAN in Automation e. V. *CANopen - Application Layer and Communication Profile*. CiA Draft Standard 301, Version 4.0.2. 2002

8 徐鹤. 车用CAN总线拓扑结构设计与性能分析方法研究. 中国农业大学硕士学位论文. 2005

9 宋晓强. CAN bus高层协议CANopen的研究以及在模块化CAN控制器上的实现. 天津大学硕士学位论文. 2004

10 王宇波. 嵌入式网络控制器的设计与实现. 天津大学硕士学位论文. 2005

11 Micro Canopen Project. http://www.microcanopen.com

12 陈涛. 汽车仪表的CANopen节点通信的研究与实现. 北京工业大学工学硕士学位论文. 2007

13 肖进军. 混合动力电动汽车CANopen总线协议的研究与实现. 北京工业大学工学硕士学位论文. 2007

14 陈骥. 基于CANopen高级协议和ED调度算法的电动汽车网络协议研究. 天津大学硕士学位论文. 2003

15 Robert Bosch GmbH. *CAN Specification Version 2.0*. September 1991

16 O. Pfeiffer, A. Ayre and C. Keydel. *Embedded Networking with CAN and CANopen*. RTC Books, San Clemente, CA, 2003

17 M. A. Livani, J. Kaiser and W. J. Jia. Scheduling hard and soft real-time communication in the controller area network. *Proceedings of the 23rd*

IFAC/IFIP Workshop on Real-Time Programming. 1998

18 Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) access method and physical layer specifications. *IEEE std 802.3-2005*. 2005: 1-2

19 FL Lian, J.R. Moyne and D.M. Tilbury. Performance evaluation of control networks: Ethernet, ControlNet, and DeviceNet. *IEEE Control Systems Magazine*. 2001, 21: 66-83

20 *SAE J1939, Joint SAE/TMC Electronic Data Interchange between Microcomputer Systems in Heavy-Duty Vehicle Applications*. SAE Standard. http://www.sae.org

21 *DeviceNet Specifications, 2nd edition*. Boca Raton, FL: Open DeviceNet Vendors Association, 1997

22 CAN in Automation e. V. *CANopen - Framework for CANopen Managers and Programmable CANopen Devices*. CiA Draft Standard Proposal 302, Version 3.1.2. 2002

23 CAN in Automation e. V. *CANopen - Device Profile for Generic I/O Modules*. CiA Draft Standard 401, Version 2.1. 2002

24 闫士珍，徐喆，宋威. 基于散列表的CANopen对象字典的设计. 计算机工程, 2009, 35(10)

25 *The CanFestival CANOpen stack manual*. Can Festival OpenSource Project. 2007. http://sourceforge.net/projects/canfestival

26 R. M. Fosler. *A CANopen Stack for PIC18 ECAN Microcontrollers*. Microchip Technology Inc., 2005. http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1824&appnote=en020605

27 J. Krakora, P. Pisa, F. Vacek, etc. *WP7 – Communication Components, D7.4 V2*. Ocera, 2004. http://www.ocera.org/download/documents/documentation/wp7.html

28 严蔚敏，吴伟民. 数据结构. 清华大学出版社, 1997

29 M. A. Weiss. *Data Structures and Algorithm Analysis in C++*. The Benjamin/Cummings Publishing Company, 1994: 181~207, 211~251

30 K. Kaspersky. 代码优化：有效使用内存. 电子工业出版社, 2004

31 R. V. Chacko, Z. V. Lakaparampil and Chandrasekar.V. CAN Based Distributed Real Time Controller Implementation for Hybrid Electric Vehicle. *Proceedings of the 2005 IEEE Conference on Vehicle Power and Propulsion*. 2005: 247-251

32 J. Barreiros, E. Costa and J. Fonseca. Jitter Reduction in a Real-Time Message

Transmission System Using Genetic Algorithms. *Proceedings of the 2000 Congress on Evolutionary Computation*. 2000: 1095-1101

33 J. Wang, B. Xu and Q. Wang. Analysis and Optimization of Message Scheduling Based on the Canopen Protocol. *Proceedings of the 25th Chinese Control Conference*. 2006: 1815-1819

34 G. Cena and A. Valenzano. Efficient polling of devices in CANopen networks. *Proceedings of IEEE Conferencon Emerging Technologies and Factory Automation 2003*. 2003: 123-130

35 Beckhoff TwinCAT. Beckhoff. 2005. http://www.beckhoff.com/english.asp?twincat/default.htm

36 *Beckhoff PC Fieldbuscard PCI CANopen FC5101 and FC5102*. Beckhoff, 2002

37 陈广涛，戴胜华. Windows CE. NET实时性能的测试与研究. 微计算机应用. 2006, 27(6): 735-738

38 L. Abeni, A. Goel, C. Krasic, etc. A Measurement-Based Analysis of the Real-Time Performance of Linux. *Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2002: 133- 142

39 范海涛，王树民. 基于RTAI的uClinux硬实时性能的实现. 电力自动化设备. 2006, 26(3): 66-68, 72

40 刘淼，王田苗，魏洪兴，等. 基于uCOS-II的嵌入式数控系统实时性分析. 计算机工程. 2006, 22(11): 222-224, 226

41 J. J. Labrosse. *MicroC/OS-II the Real-Time Kernel*. CMP Book, 2002: 73-144

42 宋威，方穗明，张明杰，等. 任务调度在CANopen主站设计中的应用. 计算机测量与控制. 2008, 16(4)

43 C. M. Krishna and YH Lee. Scanning the Issue - Special Issue on Real-Time Systems. *Proceedings of the IEEE*. 2003, 91(7): 983-985

44 W. Stallings. 操作系统——内核与设计原理. 电子工业出版社, 2004: 77~108, 145~189, 293~351

45 W. Song, S. Yan, Z. Xu and S. Fang. Transplantable CANopen Master Based on Non-preemptive Task Scheduler. *Proceedings of the 1st IEEE International Conference on Automation and Logistics 2007*, 2007: 557-562

46 C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Enviornment. *Journal of the Association for Computing Machinery*. 1973, 20(1): 46-61

47 L. George, N. Rivierre and M. Spuri. Preemptive and Non-Preemptive Real-Time Uni-Processor Scheduling. *Technical Report RR-2966, INRIA:*

Institut National de Recherche en Informatique et en Automatique. 1996

48 R. Love. *Linux Kernel Development*. Pearson Education, 2005

49 S. Molloy and P. Honeyman. Scalable Linux Scheduling. *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*. 2001: 285-296

50 B. M.E. Moret and H. D.Shapiro. *Algorithms from P to NP Volume I*. the Benjamin/Cummings Publishing Company, Inc., 1991

51 D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly, 2000: 96-148, 258-302.

52 北京华控技术有限责任公司. HK-CAN30B说明书. 2005. http://www.huakong.com.cn/products-show.asp?column_id=58&column_cat_id=27

53 J. Richter. Windows核心编程. 机械工业出版社, 2005

54 闫士珍, 徐喆, 宋威, 张卓. 基于MC9S12DP512和μC/OS-II的CANopen主站开发. 计算机工程与科学. 2008, 30

附 录

附录 1 初始对象字典

主索引	子索引	数据类型	默认初始值	描述
0x1000	0x00	32 比特整型	0xFFFF012E	设备类型，适用 DSP302 协议
0x1001	0x00	8 比特整型	0x00	错误寄存器
0x1005	0x00	32 比特整型	0x40000080	SYNC 报文 COB-ID
0x1006	0x00	32 比特整型	0x00000000	同步周期
0x1007	0x00	32 比特整型	0x00000000	同步窗口长度
0x1016	0x00	数组	0x7F	接收心跳报文周期数组
0x1016	0x01~0x7F	32 比特整型	0x00000000	各节点心跳报文周期
0x1028	0x00	数组	0x7F	紧急报文接收数组
0x1028	0x01~0x7F	32 比特整型	0x00000000	接收到的紧急事件对象
0x1200	0x00	数组	0x02	服务器端 SDO1
0x1200	0x01	32 比特整型	0x00000600	接收 SDO COB-ID
0x1200	0x02	32 比特整型	0x00000580	发送 SDO COB-ID
0x1F80	0x00	32 比特整型	0x00000007	节点启动设置。
0x1F81	0x00	数组	0x7F	从节点信息记录
0x1F81	0x01~0x7F	32 比特整型	0x00000000	从节点配置
0x1F82	0x00	数组	0x80	从节点 NMT 命令请求通道
0x1F82	0x01~0x7F	8 比特整型	0x00	从节点 NMT 命令请求
0x1F82	0x80	8 比特整型	0x00	对所有节点的命令请求
0x1F83	0x00	数组	0x80	从节点节点保护请求通道
0x1F83	0x01~0x7F	8 比特整型	0x00	从节点节点保护请求
0x1F83	0x80	8 比特整型	0x00	对所有节点的节点保护请求
0x1F84	0x00	数组	0x7F	从节点节点类型记录数组
0x1F84	0x01~0x7F	32 比特整型	0x00000000	从节点节点类型
0x1F85	0x00	数组	0x7F	从节点制造商标识记录数组
0x1F85	0x01~0x7F	32 比特整型	0x00000000	从节点制造商标识
0x1F86	0x00	数组	0x7F	从节点产品号记录数组
0x1F86	0x01~0x7F	32 比特整型	0x00000000	从节点产品号
0x1F87	0x00	数组	0x7F	从节点版本号记录数组
0x1F87	0x01~0x7F	32 比特整型	0x00000000	从节点版本号标识
0x1F88	0x00	数组	0x7F	从节点序列号记录数组
0x1F88	0x01~0x7F	32 比特整型	0x00000000	从节点序列号标识
0x1F89	0x00	32 比特整型	0x000003E8	节点启动过程等待时间

附录 2 新建从节点所需要的对象字典项

主索引	子索引	数据类型	默认初始值	描述
0x1028	nodeID	32 比特整型	0x80+nodeID	紧急事件对象
0x1280+nodeID	0x00	数组	0x03	SDO 通讯配置
0x1280+nodeID	0x01	32 比特整型	0x600+nodeID	发送 SDO COB-ID
0x1280+nodeID	0x02	32 比特整型	0x580+nodeID	接收 SDO COB-ID
0x1280+nodeID	0x03	8 比特整型	nodeID	SDO 服务端节点号
0x1F81	nodeID	32 比特整型		从节点配置
0x1F82	nodeID	8 比特整型	0x00	从节点 NMT 命令请求
0x1F83	nodeID	8 比特整型		从节点节点保护请求
0x1F84	nodeID	32 比特整型		从节点节点类型
0x1F85	nodeID	32 比特整型		从节点制造商标识
0x1F86	nodeID	32 比特整型		从节点产品号
0x1F87	nodeID	32 比特整型		从节点版本号标识
0x1F88	nodeID	32 比特整型		从节点序列号标识
0x1F89	nodeID	32 比特整型		从节点配置

附录 3 对象字典散列算法搜索时间计算函数

计算搜索时间函数 **hs_findmin()**

```
function [minr,mins,coord,percent] = hs_findmin(ODIndex)

% find the (r,s) which achieve the minimal searching time

coord = [];             % the xyz co-ordination in reslut mesh
N = 9;                  % the width of OD hash table

OD = zeros(2^N,1);      % the virtual Object Dictionary

for r = 1:2000          % choose r from 1 to 2000
    for s = 0:23
        % fill the OD
        for i = 1:length(ODIndex(:,1))
            h_addr = mod(bitshift((ODIndex(i,1)*r),-s)+ODIndex(i,2),2^N);
            OD(h_addr+1) = OD(h_addr+1) + 1;
        end

        % calculate the average search time
        p = length(find(OD))/(2^N);        % filled rate
        OD = ((ones(2^N,1) + OD).*OD)/2;
        coord = [coord; r,s,sum(OD),p];

        OD = zeros(2^N,1);
    end
end

[minS, minS_place] = min(coord(:,3));
minr = coord(minS_place,1);
mins = coord(minS_place,2);
percent = coord(minS_place,4);
```

附录 4 动态内存分配的简单实现

```
// 一个内存分配块
typedef struct _MemNode{
      int size;                          // 内存块大小
      struct _MemNode * Next;            // 内存块链表
} MemNode, * pMemNode;


// 内存分配页面
typedef struct _Page{
      int biggest;                       // 当前最大可分配空间
      pMemNode emptySpace;               // 未分配内存块链表，该链表按照内存块首地址排序
      pMemNode allocedSpace;             // 已分配内存块链表
      char d[PAGE_SIZE];                 // 实际内存空间
} Page, *pPage;

// 内存分配函数
void * cmalloc( int size)
{
      // 获得实际需要空间
      size = (size + sizeof(MemNode));
      size = (size/4 + (size&0x3 != 0))*4;

      // 是否可分配
      if (size > biggest) return NULL;

      // 在未分配内存中寻找一个可用空间
      pMem = NULL;                       // 分配的内存块指针
      pNode = Page.emptySpace;           // 准备搜索未分配内存块链表
     while(pMem != NULL)
     {
            if (pNode.size >= size) {    // 找到合适内存块
                  pMem = pNode;
                  将 pNode 从 emptySpace 链表中删除;
                  if(pNode.size - size > sizeof(MemNode)) {
                        // 该空闲内存块没有被完全使用

                        // 获得新空闲内存块地址
                        pNode = (pMemNode)((int)(pNode) + size + sizeof(MemNode));

                        // 重新计算空闲空间大小
                        pNode.size = pMem.size - size - sizeof(MemNode);

                        将 pNode 按照首地址顺序放入 emptySpace 链表;
```

```
            }
            // 计算分配空间大小
            pMem.size = (int)(pMem) - (int)(pNode) - sizeof(MemNode);
            将 pMem 放入 allocedSpace 链表;
        }
        else pNode = pNode.next;
    }

    遍历 emptySpace 重新获得 biggest;

    // 返回真正可用的首地址
    return((void *)((int)(pMem) + sizeof(MemNode)));
}

// 内存释放函数
void cfree( void * pMem)
{
    // 获得内存块地址
    mNode = (pMemNode)(((int)(pMem) - sizeof(MemNode));

    // 在 allocedSpace 中搜索该内存块
    pNode = Page.allocedSpace;
    while(pNode != mNode)    pNode = pNode.next;

    将 pNode 从 allocedSpace 链表中删除;

    在 emptySpace 链表中按照地址顺序寻找 pNode 的位置;

    if(pNode 和上一个空闲空间连续) {
        将 pNode 和上一个连续空间合并;
        用 pNode 取代上一个连续空间;
    } else if (pNode 和下一个空闲空间连续) {
        将 pNode 和下一个连续空间合并;
        用 pNode 取代下一个连续空间;
    } else  将 pNode 插入该位置;

    遍历 emptySpace 重新获得 biggest;

    return;
}
```

攻读硕士学位期间所发表的学术论文

[1] Wei Song, Shizhen Yan, Zhe Xu and Suiming Fang. Transplantable CANopen Master Based on Non-preemptive Task Scheduler. *Proceedings of the 1st IEEE International Conference on Automation and Logistics*. 2007: 557-562

[2] 宋威, 方穗明, 张明杰, 徐喆. 任务调度在 CANopen 主站设计中的应用. 计算机测量与控制. 2008, 16(4): 558-560

[3] 宋威, 方穗明, 姚丹, 张立超, 钱程. 多 FPGA 设计的时钟同步. 计算机工程. 2008, 34(7): 245-247

[4] 宋威, 方穗明. 基于 BUFGMUX 与 DCM 的 FPGA 时钟电路设计. 现代电子技术. 2006, 29(2): 141-143

[5] 徐喆, 闫士珍, 宋威, 余春暄, 段建民, 张明杰. 一种实现 CANopen 主站的方法. 发明专利, 通过初审

[6] 徐喆, 闫士珍, 宋威. 基于散列表的 CANopen 对象字典设计. 计算机工程. 2009, 35(10)

[7] 张明杰, 余春暄, 宋威. CANopen 协议栈的实现与 MySQL 数据库的结合. 计算机应用研究. 2008, 25(z)

[8] 闫士珍, 徐喆, 宋威, 张卓. 基于 MC9S12DP512 和 μC/OS-II 的 CANopen 主站开发. 计算机工程与科学. 2008, 30

致　　谢

首先，我要感谢我的导师方穗明老师。从本科教授我们数字信号处理课程，到指导我的本科毕业设计，最终到我的硕士课题，是方老师带我走进了嵌入式系统的研究领域，让我找到了今后发展的方向。三年来，无论在课题研究、工程实践、课程学习还是日常生活，方老师总是能给与我指导与鼓励。没有方老师的无私教诲与帮助，我绝不可能顺利完成课题。

我也要感谢电子信息与控制工程学院自动化专业检测方向的所有老师。段建民老师的课题给予了我们研究汽车控制系统的机会。徐喆和余春暄老师的耐心指导造就了 CANopen 研究项目的成长。还有吴晴、綦惠和陈双叶老师，他们对 CANopen 课题的支持给予了我们良好的研究环境。

当然，我也要感谢课题组其它同学的帮助。其中，肖进军学长提供了基于 DSP 的 CANopen 模型并在调度机、互斥等算法上提出了宝贵意见。陈涛学长提供了 CanFestival 模型。闫士珍同学帮我验证了对象字典的散列表算法。张明杰同学帮我实现了 CANopen 主站的 GUI 界面。张卓同学提供了主站对象字典的初始定义。宗立志同学帮助我完成了 CAN 通讯实验，并和郝亚川同学一起提供了 DSP 的实验平台。焦圣伟同学接手继续 CANopen 主站的研究工作。

此外，我还要感谢以下同学在研究生阶段中对我的帮助：贺陈、王占仓、王玲、任奎、程怀玉、李振宇、陈静、刘经纬、张红卫、孙章固和李清磊。

最后，要感谢我的父母多年来对我的教育和支持。