

工业以太网现场总线EtherCAT 驱动程序设计及应用

郇极 刘艳强 编著



北京航空航天大学出版社

策划编辑：李文铁
封面设计：run sign...

上架建议：工业自动化/数控控制

ISBN 978-7-5124-0007-8



9 787512 400078 >

定价：38.00元



工业以太网现场总线 EtherCAT 驱动程序设计及应用

郇 极 刘艳强 编著

北京航空航天大学出版社

内容简介

EtherCAT 是一种应用于工厂自动化和流程自动化领域的实时工业以太网现场总线协议,是工业通信网络国际标准 IEC61158 和 IEC61784 的组成部分。本书介绍了:实时工业以太网技术进展、EtherCAT 系统组成原理、EtherCAT 协议、从站专用集成电路芯片 ET1100、ET1100 从站硬件设计实例、EtherCAT 用于伺服驱动器控制应用协议 CoE 和 SoE、Windows XP 操作系统下 EtherCAT 主站驱动程序设计、基于微处理器的 EtherCAT 从站驱动程序设计和开发实例。

本书可作为工业自动化和计算机控制专业研究生教材或教学参考书,亦可作为 EtherCAT 协议开发技术人员的工具书。

图书在版编目(CIP)数据

工业以太网现场总线 EtherCAT 驱动程序设计及应用/
邹极等编著. -北京:北京航空航天大学出版社,2010. 3

ISBN 978 - 7 - 5124 - 0007 - 8

I. 工… II. 邹… III. 工业企业—以太网络—
总线—程序设计 IV. ①TP393. 18

中国版本图书馆 CIP 数据核字(2010)第 010428 号

工业以太网现场总线 EtherCAT 驱动程序设计及应用

邹 极 刘艳强 编著

责任编辑 李文轶

*

北京航空航天大学出版社出版发行

北京市海淀区学院路 37 号(100191) 发行部电话:(010)82317024 传真:(010)82328026

<http://www.buaapress.com.cn> E-mail:bhpress@263.net

北京市松源印刷有限公司印装 各地书店经销

开本:787×1092 1/16 印张:14.25 字数:365 千字

2010 年 3 月第 1 版 2010 年 3 月第 1 次印刷 印数:2 500 册

ISBN 978 - 7 - 5124 - 0007 - 8 定价:38.00 元

前　　言

现场总线在连接数字伺服、传感器以及 PLC-IO 等设备的控制系统中已经获得广泛应用，实时工业以太网(RTE, Real Time Ethernet)是当前现场总线技术的一个重要发展方向。目前，国际上有多种实时工业以太网协议，国际电工委员会(IEC, International Electro Technical Commission)制定了两个与实时工业以太网相关的标准。

(1) IEC61158^[1-6]: 工业通信网络——现场总线规范(Industrial communication networks-Fieldbus specifications);

(2) IEC61784^[7]: 工业通信网络——行规规范(Industrial communication networks-Profiles)。

EtherCAT(Ethernet for Control Automation Technology)^[8,9]是一种基于以太网的实时工业现场总线通信协议和国际标准。它具有高速和高数据有效率的特点，支持多种物理拓扑结构。其主站使用标准的以太网通信控制器，从站使用专用的从站控制芯片。

EtherCAT由德国BECKHOFF自动化公司于2003年提出，并于2007年12月成为国际标准，是IEC61158和IEC61784中定义的第十二种通信协议标准。

虽然国际标准的颁布已有一段时间，国外采用EtherCAT技术的自动化设备也已经开始进入国内，但国内对EtherCAT产品与技术的开发和应用尚处于起步阶段。为了支持EtherCAT技术在国内的应用与发展，有必要对其系统原理、协议内容，特别是软、硬件设计方法，进行系统的全面的介绍。

本书的章节安排如下：

第1章为EtherCAT概述，简要介绍实时工业以太网的技术进展和EtherCAT系统组成原理。

第2章介绍EtherCAT协议，主要内容包括：系统组成、数据帧结构、报文寻址、通信服务、分布式时钟、通信模式、EtherCAT接口初始化以及应用层协议等。

第3章介绍实现EtherCAT数据链路层协议的专用集成电路芯片及其基本功能，着重介绍了BECKHOFF公司的ET1100^[10,11]芯片。

第4章介绍EtherCAT硬件设计，给出了微处理器操作ET1100的EtherCAT从站和直接I/O控制EtherCAT从站的硬件设计实例。

第5章介绍了EtherCAT用于伺服驱动器控制应用协议，包括CoE和SoE两种协议形式，还着重介绍周期性过程数据通信和非周期性数据通信的报文格式。

第6章介绍Windows XP操作系统下EtherCAT主站驱动程序设计，着重介绍系统初始化和周期性数据传输的C++程序实现，给出了关键的程序流程图和主要程序源代码。

第7章介绍基于微处理器的EtherCAT从站驱动程序设计，除了给出基本的程序框架以外，还着重介绍EtherCAT接口初始化和周期性数据处理的程序实现方法。

本书第2章、第3章和第5章的内容是对国际标准IEC61158、IEC61800和德国BECKHOFF自动化有限公司的ET1100芯片手册等众多文献的整理与汇编；通过作者的理解，添加

了一些图、表，使其说明更清晰准确。此外，书中作者还对一些参考文献中不完全的数据进行了分析和测定，对一些说明、术语作了翻译和一致性处理，并设计了本书的章节顺序。本书介绍的硬件设计实例和驱动程序示例都是基于作者多年的理论知识和开发经验，并对其部分原理图和程序源代码作了必要的组织和整理。

本书可作为工业自动化和计算机控制专业研究生教材或教学参考书，亦可作为 EtherCAT 协议开发技术人员的工具书。

在本书的撰写过程中，力求体系合理，文理清楚，概念准确，用词规范。但由于作者水平所限，对于书中疏漏及不妥之处，欢迎广大读者予以批评指正。

作 者

2009 年 8 月于北京

目 录

第 1 章 概 述	1
1.1 实时工业以太网概述	1
1.2 EtherCAT 协议概述	3
第 2 章 EtherCAT 协议	5
2.1 EtherCAT 系统组成	5
2.1.1 EtherCAT 主站组成	5
2.1.2 EtherCAT 从站组成	6
2.1.3 EtherCAT 物理拓扑结构	8
2.2 EtherCAT 数据帧结构	9
2.3 EtherCAT 报文寻址和通信服务	10
2.3.1 EtherCAT 网段寻址	11
2.3.2 设备寻址	12
2.3.3 逻辑寻址和 FMMU	14
2.3.4 通信服务和 WKC	16
2.4 分布时钟	17
2.4.1 分布时钟描述	17
2.4.2 传输延时和时钟初始偏移量的测量	18
2.4.3 时钟同步	19
2.5 通信模式	21
2.5.1 周期性过程数据通信	21
2.5.2 非周期性邮箱数据通信	24
2.6 状态机和通信初始化	25
2.7 应用层协议	27
第 3 章 EtherCAT 从站控制芯片	29
3.1 ESC 概述	30
3.1.1 ESC 芯片种类	30
3.1.2 ESC 存储空间	30
3.1.3 ESC 特征信息	35
3.2 ESC 芯片 ET1100	37
3.2.1 ET1100 引脚定义	37
3.2.2 物理通信端口	41

3.2.3 PDI 接口	45
3.2.4 配置引脚	52
3.2.5 其他引脚	53
3.3 ESC 数据链路控制	55
3.3.1 ESC 数据帧处理	55
3.3.2 ESC 通信端口控制	56
3.3.3 数据链路错误检测	58
3.3.4 ESC 数据链路地址	59
3.3.5 逻辑寻址控制	60
3.4 ESC 应用层控制	61
3.4.1 状态机控制和状态	61
3.4.2 中断控制	63
3.4.3 看门狗控制	64
3.5 存储同步管理	65
3.5.1 存储同步管理器概述	65
3.5.2 缓存类型数据交换	67
3.5.3 邮箱数据通信机制	68
3.6 从站信息接口	71
3.6.1 EEPROM 内容	71
3.6.2 EEPROM 访问控制	72
3.6.3 EEPROM 操作错误处理	75
3.7 分布时钟操作	76
3.7.1 分布时钟信号	76
3.7.2 分布时钟的初始化	80
3.7.3 同步信号的配置	83
第 4 章 EtherCAT 硬件设计	84
4.1 EtherCAT 从站 PHY 器件选择	84
4.2 微处理器操作的 EtherCAT 从站硬件设计实例	85
4.2.1 ET1100 的接线	86
4.2.2 ET1100 配置电路	88
4.2.3 MII 接线	89
4.2.4 微处理器接口引脚接线	91
4.3 直接 I/O 控制 EtherCAT 从站硬件设计实例	92
第 5 章 EtherCAT 伺服驱动器控制应用协议	95
5.1 CoE(CANopen over EtherCAT)	95
5.1.1 CoE 对象字典	96
5.1.2 周期性过程数据通信	97

5.1.3 CoE 非周期性数据通信	98
5.1.4 应用层行规	108
5.2 SoE(SERCOS over EtherCAT)	115
5.2.1 SoE 状态机	115
5.2.2 IDN 继承	116
5.2.3 SoE 过程数据映射	117
5.2.4 SoE 服务通道	120
第 6 章 EtherCAT 主站驱动程序	127
6.1 数据定义头文件	128
6.2 网卡操作相关类的定义和实现	135
6.2.1 基于 NDIS 的网卡驱动程序	135
6.2.2 CEcNpfDevice 类	137
6.2.3 CNpfInfo 类	140
6.2.4 获得计算机网卡信息	141
6.2.5 打开网卡	143
6.2.6 发送数据帧	146
6.2.7 接收数据帧	147
6.2.8 关闭网卡	151
6.3 从站设备对象的定义和实现	152
6.3.1 CEcSimSlave 类的定义	153
6.3.2 CEcSimSlave 类的实现	154
6.4 主站设备对象的定义和实现	157
6.4.1 CEcSimMaster 类的定义	157
6.4.2 初始化和启动 CEcSimMaster 数据对象	159
6.4.3 配置从站设备对象	160
6.4.4 状态机运行	163
6.4.5 发送非周期性 EtherCAT 数据报文	166
6.4.6 发送周期性 EtherCAT 数据帧	168
6.4.7 接收 EtherCAT 数据帧	174
6.5 主站实例程序	177
6.5.1 通信配置初始化流程	178
6.5.2 周期性运行控制	181
第 7 章 从站驱动程序	183
7.1 从站驱动程序头文件 ec_def.h	183
7.2 从站基本操作	192
7.3 从站驱动程序总体结构	195
7.4 从站周期性数据的处理	198

7.4.1 同步运行模式	198
7.4.2 自由运行模式	199
7.5 从站非周期性事件的处理	201
7.6 从站状态机的处理	201
7.6.1 状态机处理流程	202
7.6.2 检查 SM 通道设置	206
7.6.3 启动邮箱数据通信	212
7.6.4 启动周期性输入数据通信	213
7.6.5 启动周期性输出数据通信	215
7.6.6 停止 EtherCAT 数据通信	216
参考文献	218

工业以太网现场总线 EtherCAT 驱动程序设计及应用

郇 极 刘艳强 编著

北京航空航天大学出版社

内容简介

EtherCAT 是一种应用于工厂自动化和流程自动化领域的实时工业以太网现场总线协议,是工业通信网络国际标准 IEC61158 和 IEC61784 的组成部分。本书介绍了:实时工业以太网技术进展、EtherCAT 系统组成原理、EtherCAT 协议、从站专用集成电路芯片 ET1100、ET1100 从站硬件设计实例、EtherCAT 用于伺服驱动器控制应用协议 CoE 和 SoE、Windows XP 操作系统下 EtherCAT 主站驱动程序设计、基于微处理器的 EtherCAT 从站驱动程序设计和开发实例。

本书可作为工业自动化和计算机控制专业研究生教材或教学参考书,亦可作为 EtherCAT 协议开发技术人员的工具书。

图书在版编目(CIP)数据

工业以太网现场总线 EtherCAT 驱动程序设计及应用/

郇极等编著.-北京:北京航空航天大学出版社,2010.3

ISBN 978 - 7 - 5124 - 0007 - 8

I. 工… II. 郞… III. 工业企业—以太网络—
总线—程序设计 IV. ①TP393.18

中国版本图书馆 CIP 数据核字(2010)第 010428 号

工业以太网现场总线 EtherCAT 驱动程序设计及应用

郇 极 刘艳强 编著

责任编辑 李文轶

*

北京航空航天大学出版社出版发行

北京市海淀区学院路 37 号(100191) 发行部电话:(010)82317024 传真:(010)82328026

<http://www.buaapress.com.cn> E-mail: bhpress@263.net

北京市松源印刷有限公司印装 各地书店经销

*

开本:787×1092 1/16 印张:14.25 字数:365 千字

2010 年 3 月第 1 版 2010 年 3 月第 1 次印刷 印数:2 500 册

ISBN 978 - 7 - 5124 - 0007 - 8 定价:38.00 元

第1章 概述

将计算机网络中的以太网技术应用于工业自动化领域构成的工业控制以太网，简称工业以太网或以太网现场总线，是当前工业控制现场总线技术的一个重要发展方向。与使用传统技术的现场总线相比，以太网现场总线具有以下优点：

- 传输速度快，数据包容量大，传输距离长；
- 使用通用以太网元器件，性能价格比高；
- 可以接入标准以太网网段。

1.1 实时工业以太网概述

实时以太网（RTE,Real Time Ethernet）是常规以太网技术的延伸，以便满足工业控制领域的实时性数据通信要求。目前，国际上有多种实时工业以太网协议，国际电工委员会 IEC 制定了两个与实时工业以太网相关的标准：

(1) IEC61158：工业通信网络——现场总线规范（Industrial communication networks - Fieldbus specifications）

IEC61158 是 IEC 制定的现场总线国际标准，于 1999 年发布了第一版，包括 8 种现场总线协议。随着实时以太网的发展，IEC61158 也新增了实时以太网的标准，于 2007 年 12 月出版的 IEC61158 第 4 版包括了 10 种工业以太网协议标准，如表 1.1 所列。其中 Type2 CIP (Common Industry Protocol) 包括 DeviceNet、ControlNet 现场总线和 Ethernet/IP 实时工业以太网。

(2) IEC61784：工业通信网络——行规规范（Industrial communication networks - Profiles）

IEC61784 为 IEC61158 中的现场总线标准制定了应用行规标准，其中第一部分 IEC61784-1 为传统现场总线的应用行规族（CPF，Communication Profile Family），第二部分 IEC61784-2 为基于 ISO/IEC 8802-3 的实时工业以太网 CPF。表 1.1 中也同时列出了 CPF 与 IEC61158 当中技术名称的对应关系。

以太网的介质访问控制 MAC(Media Access Control)方式采用带有冲突检测的载波侦听多路访问机制 CSMA/CD(Carrier Sense Multiple Access with Collision Detection)。这是一种非确定性的介质访问控制方式，不能满足工业现场总线的实时性要求。目前，市场上已有的实时工业以太网根据不同的实时性和成本要求使用不同的实现原理，大致可以分为以下三种类型，如图 1.1 所示。

表 1.1 IEC61158 第四版现场总线类型

类型编号	技术名称	CPF	支持组织和公司	分类
Type 1	TS61158		IEC	现场总线
Type 2	CIP	CPF2	CI(美国)、ODVA(美国)、Rockwell (美国)	DeviceNet、ControlNet 和 Ethernet/IP
Type 3	Profibus	CPF3	PI、Siemens(德)	现场总线
Type 4	P-NET	CPF4	Process Data(丹麦)	现场总线
Type 5	FF HSE	CPF1	FF、Fisher-Rosemount(美国)	高速以太网
Type 6	Swift Net	CPF7	SHIP STAR、Boeing(美国)	被撤销
Type 7	WorldFIP IP	CPF5	WorldFIP、Alstom(法国)	现场总线
Type 8	INTERBUS	CPF6	INTERBUS Club、Phoenix contact (德国)	现场总线
Type 9	FF H1	CPF1	FF(美国)	现场总线
Type 10	PROFINET	CPF3	PI、Siemens(德国)	实时以太网
Type 11	TC-net	CPF11	Toshiba(日本)	实时以太网
Type 12	EtherCAT	CPF12	ETG、Beckhoff(德国)	实时以太网
Type 13	Ethernet PowerLink	CPF13	EPSG、B&R(奥地利)	实时以太网
Type 14	EPA	CPF14	浙大中控等(中国)	实时以太网
Type 15	Modbus-RTPS	CPF15	MODBUS、IDA(美国)	实时以太网
Type 16	SERCOS I、II	CPF16	IGS(德国)	现场总线
Type 17	VNET /IP	CPF10	Yokogawa(日本)	实时以太网
Type 18	CC-LINK	CPF8	三菱(日本)	现场总线
Type 19	SERCOS III	CPF16	IGS(德国)	实时以太网
Type 20	HART	CPF9	HART 通信基金协会(美国)	现场总线

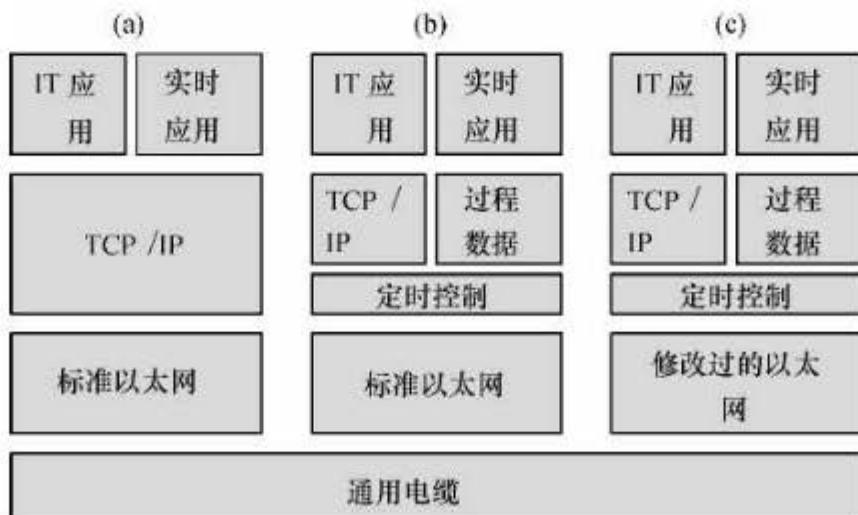


图 1.1 以太网通信模型

(1) 基于 TCP/IP 的实现 (如图 1.1 (a) 所示)

这种方式仍使用 TCP/IP 协议栈，通过上层合理的控制来应对通信中非确定性因素。此时，实时网络可以与商用网络自由地通信。常用的通信控制手段有：合理调度，减少冲突的可能性；定义数据帧的优先级，为实时数据分配最高的优先级；使用交换式以太网等。使用这种方式的典型协议有 Modbus/TCP 和 Ethernet/IP 等。这种方式不能实现很好的实时性，只适用于对实时性要求不高的工业过程自动化应用。

(2) 基于以太网的实现 (如图 1.1 (b) 所示)

这种方式仍然使用标准的、未修改的以太网通信硬件，但是不使用 TCP/IP 来传输过程数据。它引入了一种专门的过程数据传输协议，使用特定以太类型的以太网帧进行传输。TCP/IP 协议栈可以通过一个时间控制层分配一定的时间片来使用以太网资源。这类协议主要有 Ethernet Powerlink、EPA (Ethernet for Plant Automation)、PROFINet RT 等通过这种方式。这种方式可以实现较高的实时性。

(3) 修改以太网的实现 (如图 1.1 (c) 所示)

为了获得响应时间小于 1 ms 的硬实时，通过这种方式对以太网协议进行了修改。其从站由专门的硬件实现。在实时通道内由实时 MAC 接管通信控制，彻底避免报文冲突，简化通信数据处理。非实时数据仍然可以在开放通道内按照原来的协议传输。这种方式下的典型协议有 EtherCAT、SERCOS 和 III 和 PROFINet IRT 等。

1.2 EtherCAT 协议概述

EtherCAT 是由德国 BECKHOFF 自动化公司于 2003 年提出的实时工业以太网技术。它具有高速和高数据有效率的特点，支持多种设备连接拓扑结构。其从站节点使用专用的控制芯片，主站使用标准的以太网控制器。

EtherCAT 的主要特点如下：

① 广泛的适用性，任何带商用以太网控制器的控制单元都可作为 EtherCAT 主站。从小型的 16 位处理器到使用 3 GHz 处理器的 PC 系统，任何计算机都可以成为 EtherCAT 控制系统。

② 完全符合以太网标准，EtherCAT 可以与其他以太网设备及协议并存于同一总线，以太网交换机等标准结构组件也可以用于 EtherCAT。

③ 无须从属子网，复杂的节点或只有 2 位的 I/O 节点都可以用作 EtherCAT 从站；

④ 高效率，最大化利用以太网带宽进行用户数据传输。

⑤ 刷新周期短，可以达到小于 100 μs 的数据刷新周期，可以用于伺服技术中底层的闭环控制。

⑥ 同步性能好，各从站节点设备可以达到小于 1μs 的时钟同步精度。

目前，EtherCAT 已经进入多种相关国际标准：

- IEC61158 中 Type12；

- IEC61784 中 CPF12；

- IEC61800 中，EtherCAT 支持 CANopen DS402 和 SERCOS；

- ISO15745 中, EtherCAT 支持 DS301。

EtherCAT 支持多种设备连接拓扑结构: 线形、树形或星形结构, 可以选用的物理介质有 100Base-TX 标准以太网电缆或光缆。使用 100Base-TX 电缆时站间距离可以达到 100m。整个网络最多可以连接 65 535 个设备。使用快速以太网“全双工”通信技术构成主从式的环型结构如图 1.2 所示。

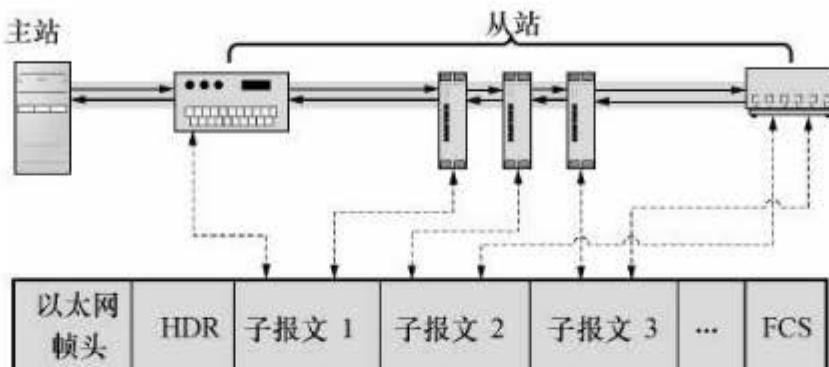


图 1.2 EtherCAT 运行原理

从以太网的角度看,一个 EtherCAT 网段可被简单地看作一个独立的以太网设备。该“设备”接收并发送以太网报文。然而,这个“设备”并没有以太网控制器及相应的微处理器,而是由多个 EtherCAT 从站组成。这些从站可直接处理接收的报文,并从报文中提取或插入相关的用户数据,然后将该报文传输到下一个 EtherCAT 从站。最后一个 EtherCAT 从站发回经过完全处理的报文,并由第一个从站作为响应报文发送给控制单元。这个过程利用了以太网设备独立处理双向传输 (Tx 和 Rx) 的特点,并运行在全双工模式下,发出的报文又通过 Rx 线返回到控制单元。

报文经过从站节点时,从站识别出相关的命令并做出相应的处理。信息的处理在硬件中完成,延迟时间约为 100~500 ns (取决于物理层器件),通信性能独立于从站设备控制微处理器的响应时间。每个从站设备有最大容量为 64 K 字节的可编址内存,可完成连续的或同步的读写操作。多个 EtherCAT 命令数据可以被嵌入到一个以太网报文中,每个数据对应独立的设备或内存区。

从站设备可以构成多种形式的分支结构,独立的设备分支可以放置于控制柜中或机器模块中,再用主线连接这些分支结构。

EtherCAT 大大提高了现场总线的性能,例如,控制 1 000 个开关量输入和输出的刷新时间为 30 μs。单个以太网帧最多可容纳 1 486 字节的过程数据,相当于 12 000 位开关量数字输入和输出,刷新时间为 300 μs。控制 100 个伺服电机的数据通信周期约为 100 μs。

EtherCAT 使用一个专门的以太网数据帧类型定义,用以太网数据帧传输 EtherCAT 数据包,也可以使用 UDP/IP 协议格式传输 EtherCAT 数据包。一个 EtherCAT 数据包可以由多个 EtherCAT 子报文组成,如图 1.2 所示。EtherCAT 从站不处理非 EtherCAT 数据帧,其他类型的以太网应用数据可以分段打包为 EtherCAT 数据子报文在网段内透明传输,以实现相应的通信服务。

第2章 EtherCAT协议

2.1 EtherCAT 系统组成

EtherCAT 是一种实时工业以太网技术，它充分利用了以太网的“全双工”特性。使用主从模式介质访问控制（MAC），主站发送以太网帧给各从站，从站从数据帧中抽取数据或将数据插入数据帧。主站使用标准的以太网接口卡，从站使用专门的 EtherCAT 从站控制器 ESC（EtherCAT Slave Controller）。EtherCAT 物理层使用标准的以太网物理层器件。

从以太网的角度来看，一个 EtherCAT 网段就是一个以太网设备，它接收和发送标准的 ISO/IEC8802—3 以太网数据帧。但是，这种以太网设备并不局限于一个以太网控制器及相应的微处理器，它可由多个 EtherCAT 从站组成，如图 2.1 所示。这些从站可以直接处理接收的报文，并从报文中提取或插入相关的用户数据，然后将该报文传输到下一个 EtherCAT 从站。最后一个 EtherCAT 从站发回经过完全处理的报文，并由第一个从站作为响应报文将其发送给控制单元。

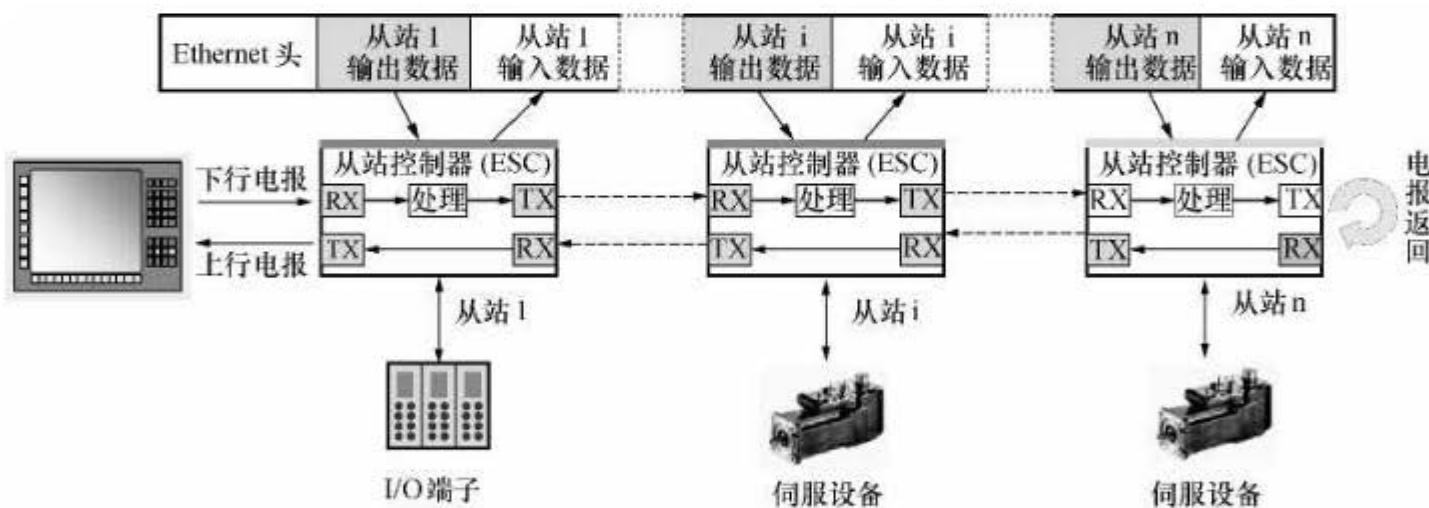


图 2.1 EtherCAT 运行原理

2.1.1 EtherCAT 主站组成

EtherCAT 主站使用标准的以太网控制器，传输介质通常使用 100BASE-TX 规范的 5 类 UTP 线缆，如图 2.2 所示。通信控制器完成以太网数据链路的介质访问控制（MAC，Media Access Control）功能，物理层芯片 PHY 实现数据编码、译码和收发，它们之间通过一个 MII（Media Independent Interface）接口交互数据。MII 是标准的以太网物理层接口，定义了与传输介质无关的标准电气和机械接口，使用这个接口将以太网数据链路层和物理

层完全隔离开，使以太网可以方便地选用任何传输介质。隔离变压器实现信号的隔离，提高通信的可靠性。

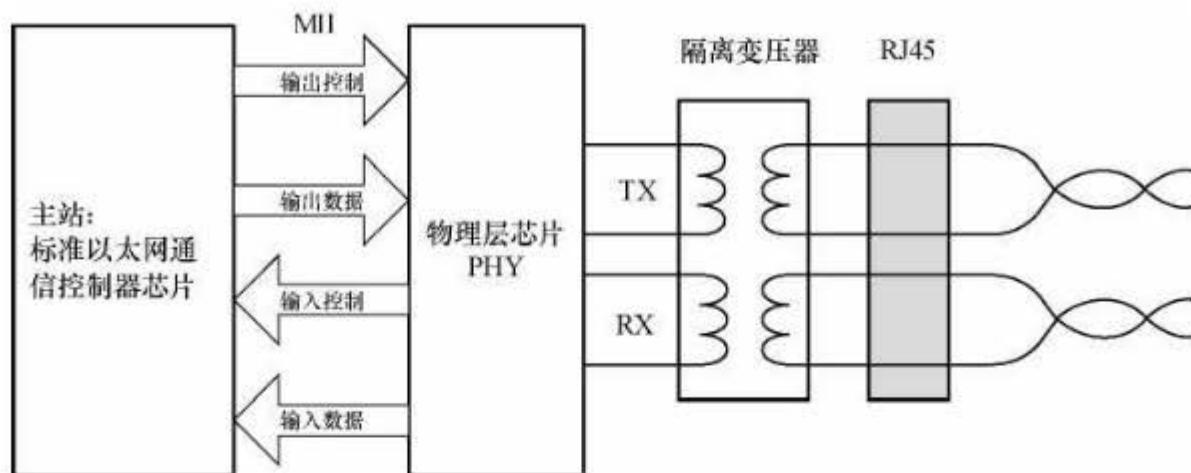


图 2.2 EtherCAT 物理层连接原理图

在基于 PC 的主站中，通常使用网络接口卡 NIC (Network Interface Card)，其中的网卡芯片集成了以太网通信控制器和物理数据收发器。而在嵌入式主站中，通信控制器通常嵌入到微处理器中。

2.1.2 EtherCAT 从站组成

EtherCAT 从站设备同时实现通信和控制应用两部分功能，其结构如图 2.3 所示，由以下四部分组成。

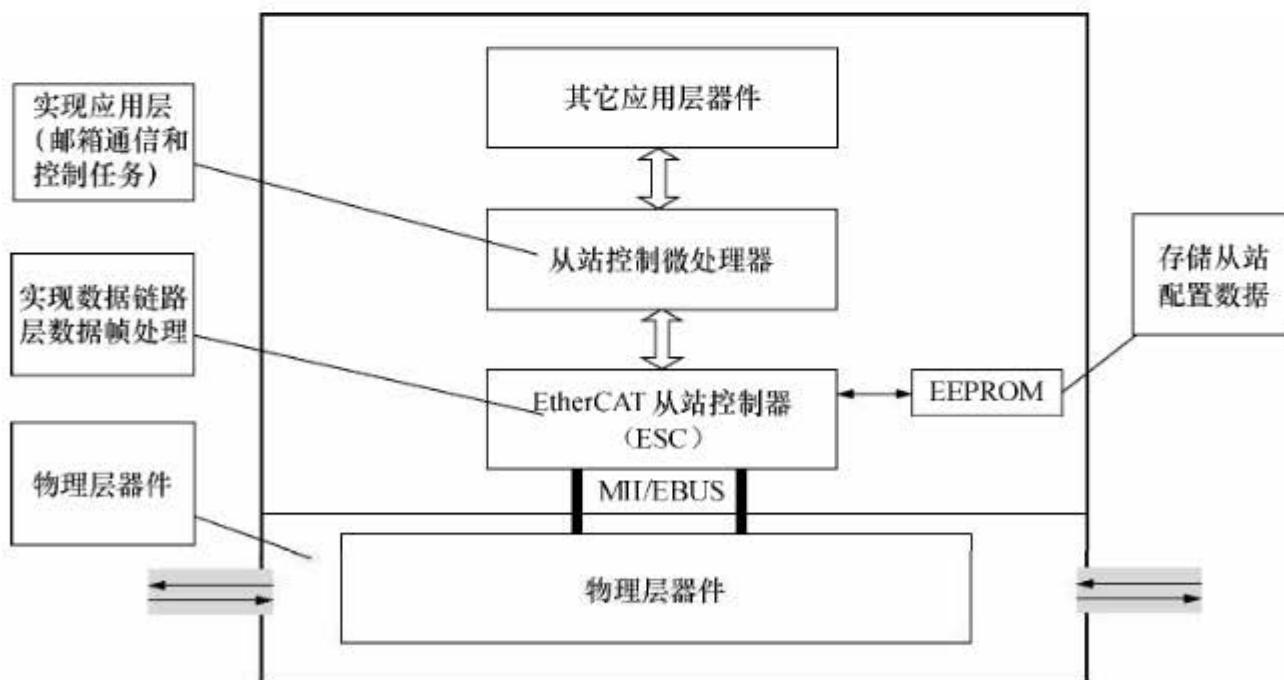


图 2.3 EtherCAT 从站组成

1. EtherCAT 从站控制器 ESC

EtherCAT 从站通信控制器芯片 ESC 负责处理 EtherCAT 数据帧，并使用双端口存储区

实现 EtherCAT 主站与从站本地应用的数据交换。各个从站 ESC 按照各自在环路上的位置顺序移位读写数据帧。在报文经过从站时，ESC 从报文中提取发送给自己的输出命令数据并将其存储到内部存储区，该输入数据从内部存储区又被写到相应的子报文中。数据的提取和插入都是由数据链路层硬件完成的。

ESC 具有四个数据收发端口，每个端口都可以收发以太网数据帧。数据帧在 ESC 内部的传输顺序是固定的，如图 2.4 所示。通常，数据从端口 0 进入 ESC，然后按照端口 3 端口 1 端口 2 端口 0 的顺序依次传输。如果 ESC 检测到某个端口没有外部链接，则自动闭合此端口，数据将自动回环并转发到下一端口。一个 EtherCAT 从站设备至少使用两个数据端口，使用多个数据端口可以构成多种物理拓扑结构。

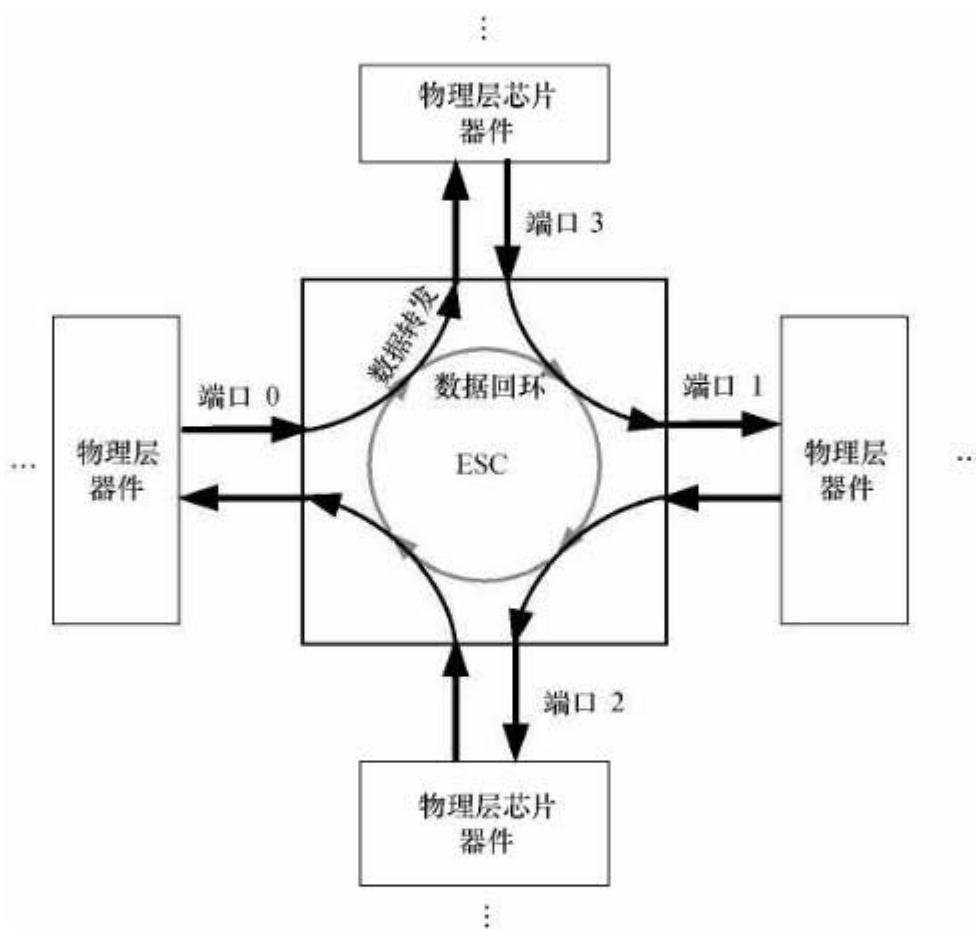


图 2.4 ESC 数据传输顺序

ESC 使用两种物理层接口模式：MII 和 EBUS。MII 是标准的以太网物理层接口，使用外部物理层芯片，一个端口的传输延时约为 500 ns。EBUS 是德国 BECKHOFF 公司使用 LVDS (Low Voltage Differential Signaling) 标准定义的数据传输标准，可以直接连接 ESC 芯片，不需要额外的物理层芯片，从而避免了物理层的附加传输延时，一个端口的传输延时约为 100ns。EBUS 最大传输距离为 10 m，适用于放置距离较近的 I/O 设备或伺服驱动器之间的连接。

2. 从站控制微处理器

微处理器负责处理 EtherCAT 通信和完成控制任务。微处理器从 ESC 读取控制数据，实现设备控制功能，并采样设备的反馈数据，写入 ESC，由主站读取。通信过程完全由 ESC 处理，与设备控制微处理器响应时间无关。从站控制微处理器性能选择取决于设备控制任

务，可以使用 8 位、16 位的单片机及 32 位的高性能处理器。

3. 物理层器件

从站使用 MII 接口时，需要使用物理层芯片 PHY 和隔离变压器等标准以太网物理层器件。使用 EBUS 时不需要任何其他芯片。

4. 其他应用层器件

针对控制对象和任务需要，微处理器可以连接其他控制器件。

2.1.3 EtherCAT 物理拓扑结构

在逻辑上，EtherCAT 网段内从站设备的布置构成一个开口的环型总线。在开口的一端，主站设备直接或者通过标准以太网交换机插入以太网数据帧，并在另一端接收经过处理的数据帧。所有的数据帧都被从第一个从站设备转发到后续的节点。最后一个从站设备将数据帧返回到主站。

EtherCAT 从站的数据帧处理机制允许在 EtherCAT 网段内的任一位置使用分支结构，同时不打破逻辑环路。分支结构可以构成各种物理拓扑(如线形、树形、星形，菊花链形)以及各种拓扑结构的组合，从而使设备连接布线非常灵活方便。如图 2.5 中，主站发出数据帧后的传输顺序如图中数字标号①~⑭所示，其中从站 8 使用了 ESC 的全部四个端口，构成星型拓扑。

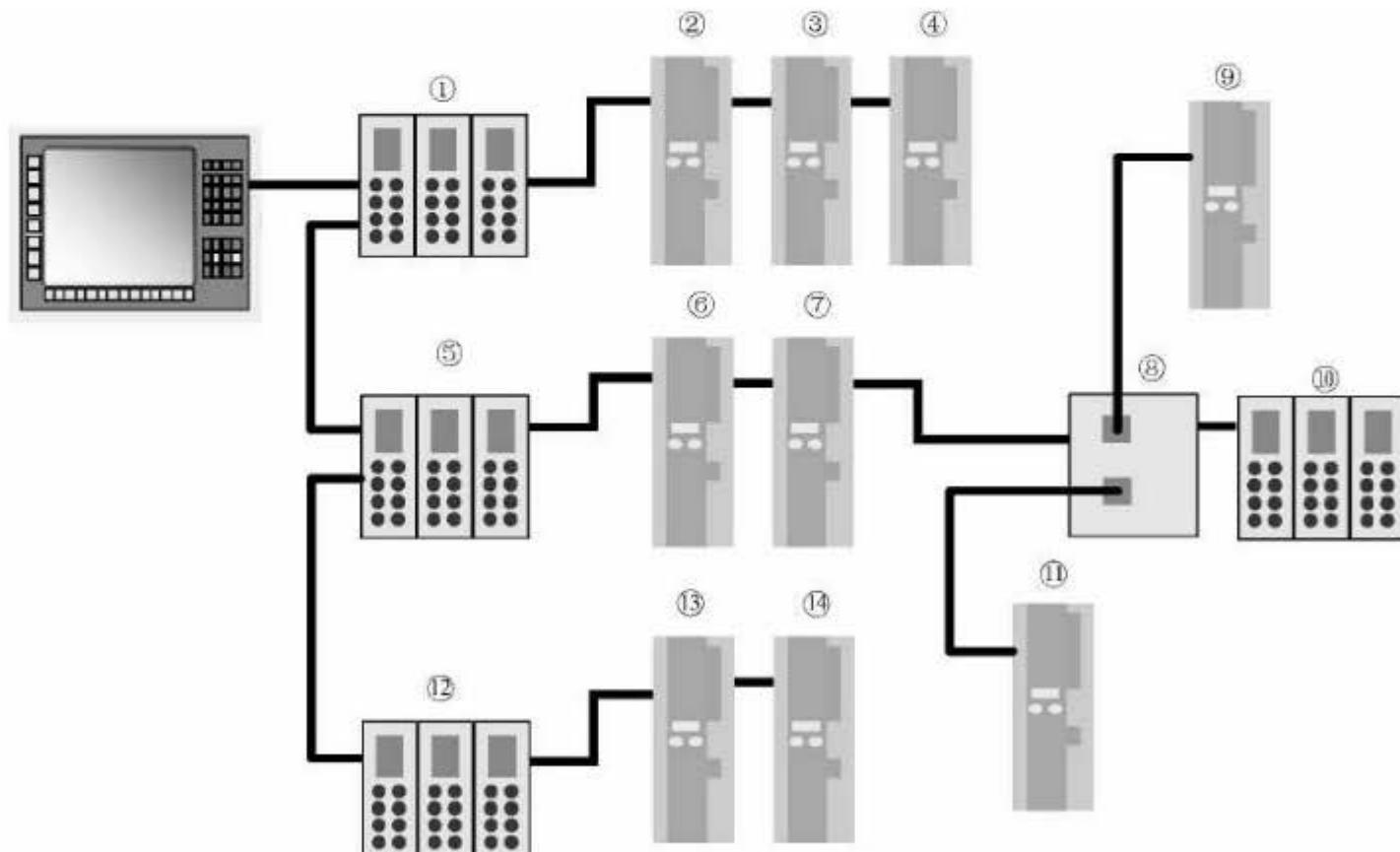


图 2.5 EtherCAT 线型拓扑结构

2.2 EtherCAT 数据帧结构

EtherCAT数据直接使用以太网数据帧传输，数据帧使用帧类型0x88A4。EtherCAT数据包括2个字节的数据头和44~1498字节的数据。数据区由一个或多个EtherCAT子报文组成，每个子报文对应独立的设备或从站存储区域，如图2.6所示。表2.1给出了EtherCAT数据帧结构定义。

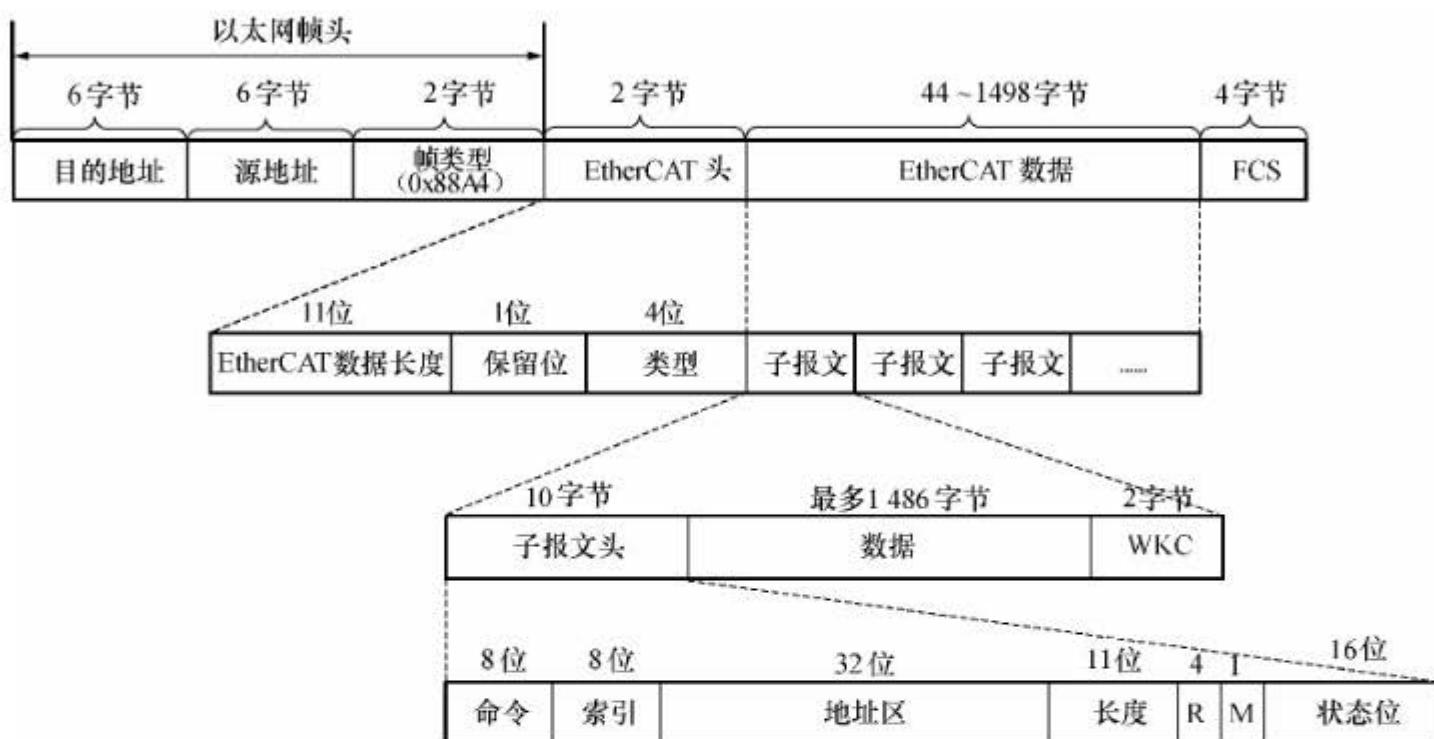


图2.6 EtherCAT报文嵌入以太网数据帧

表2.1 EtherCAT帧结构定义

名称	含义
目的地址	接收方MAC地址
源地址	发送方MAC地址
帧类型	0x88A4
EtherCAT头：长度	EtherCAT数据区长度，即所有子报文长度总和
EtherCAT头：类型	1：表示与从站通信；其余保留
FCS (Frame Check Sequence)	帧校验序列

每个EtherCAT子报文包括子报文头、数据域和相应的工作计数器（WKC，Working Counter）。WKC记录了子报文被从站操作的次数，主站为每个通信服务子报文设置预期的WKC。发送子报文中的工作计数器初值为0，子报文被从站正确处理后，工作计数器的

值将增加一个增量，主站比较返回子报文中的 WKC 和预期 WKC 来判断子报文是否被正确处理。WKC 由 ESC 在处理数据帧的同时进行处理，不同的通信服务对 WKC 的增加方式不同。表 2.2 给出了 EtherCAT 子报文的结构定义。

表 2.2 EtherCAT 子报文结构定义

名 称	含 义
命令	寻址方式及读写方式
索引	帧编码
地址区	从站地址
长度	报文数据区长度
R	保留位
M	后续报文标志
状态位	中断到来标志
数据区	子报文数据结构，用户定义
WKC	工作计数器

也可以使用 UDP/IP 协议格式传输 EtherCAT 数据，使用 UDP 端口 0x88A4，如图 2.7 所示。

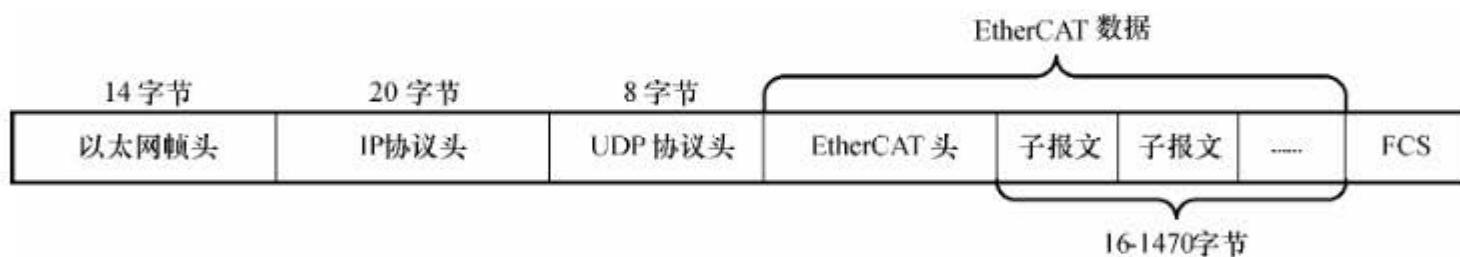


图 2.7 EtherCAT 数据帧嵌入 UDP 数据帧

2.3 EtherCAT 报文寻址和通信服务

EtherCAT 通信由主站发送 EtherCAT 数据帧读写从站设备的内部存储区来实现，EtherCAT 报文使用多种寻址方式操作 ESC 内部存储区，实现多种通信服务。

EtherCAT 网络寻址方式如图 2.8 所示。一个 EtherCAT 网段相当于一个以太网设备，主站首先使用以太网数据帧头的 MAC 地址寻址到网段，然后使用 EtherCAT 子报文头中的 32 位地址寻址到段内设备。

段内寻址可以使用两种方式：设备寻址和逻辑寻址。设备寻址针对某一个从站进行读写操作。逻辑寻址面向过程数据，可以实现多播，同一个子报文可以读写多个从站设备。支持所有寻址模式的从站称为完整型从站，而只支持部分寻址模式的从站称为基本从站。

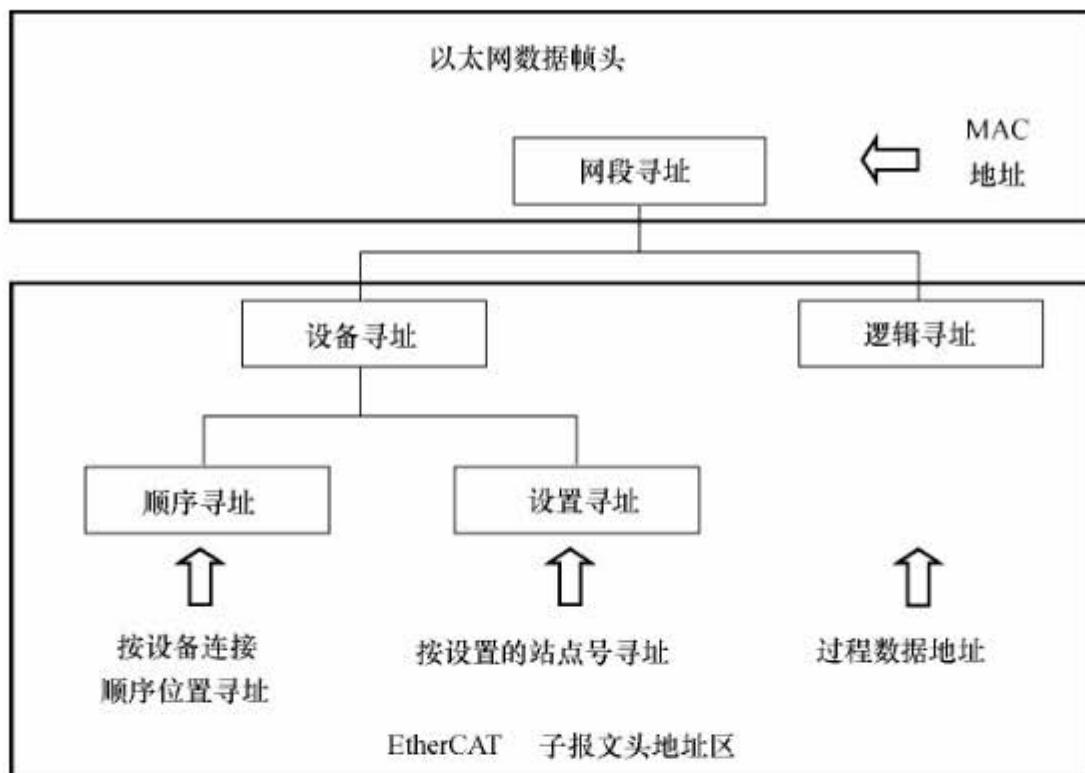


图 2.8 EtherCAT 网络寻址模式

2.3.1 EtherCAT 网段寻址

根据 EtherCAT 主站及其网段的连接方式不同，可以使用两种方式寻址到网段：

(1) 直连模式

一个 EtherCAT 网段直接连到主站设备的标准以太网端口，如图 2.9 所示。此时，主站使用广播 MAC 地址，EtherCAT 数据帧如图 2.10 所示。

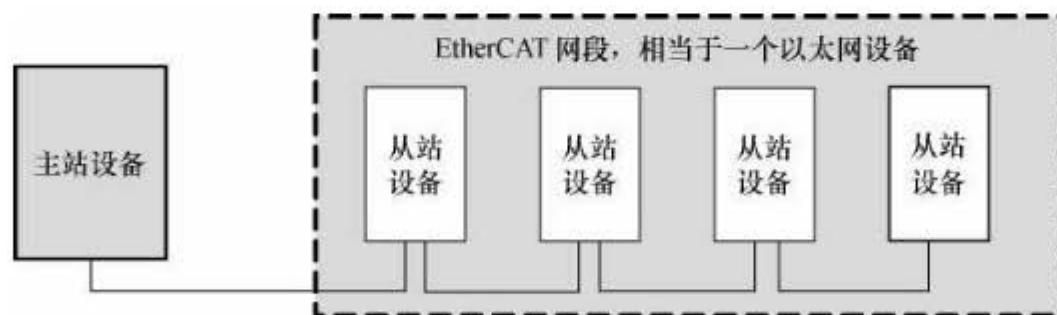


图 2.9 直连模式中的 EtherCAT 网段

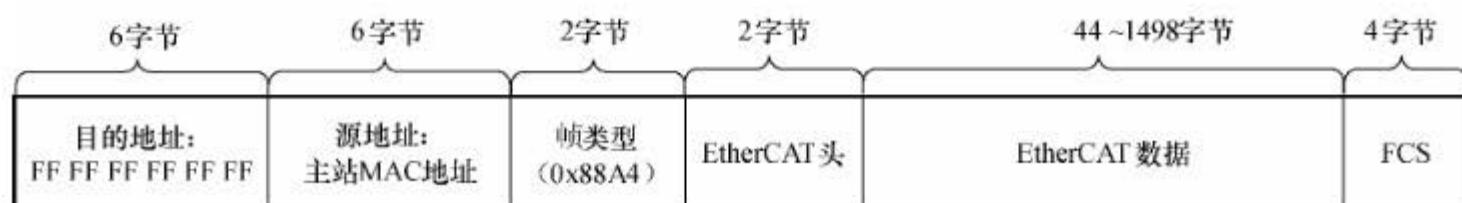


图 2.10 直连模式下 EtherCAT 网段寻址地址内容

(2) 开放模式

EtherCAT 网段连接到一个标准以太网交换机上，如图 2.11 所示。此时，一个网段需要一个 MAC 地址，主站发送的 EtherCAT 数据帧中目的地址是它所控制的网段的 MAC 地址，如图 2.12 所示。

EtherCAT 网段内的第一个从站设备有一个 ISO/IEC 8802.3 的 MAC 地址，这个地址表示了整个网段，这个从站称为段地址从站，它能够交换以太网帧内的目的地址区和源地址区。如果 EtherCAT 数据帧通过 UDP 传送，这个设备也会交换源和目的 IP 地址、以及源和目的 UDP 端口号，使响应的数据帧完全满足 UDP/IP 协议标准。

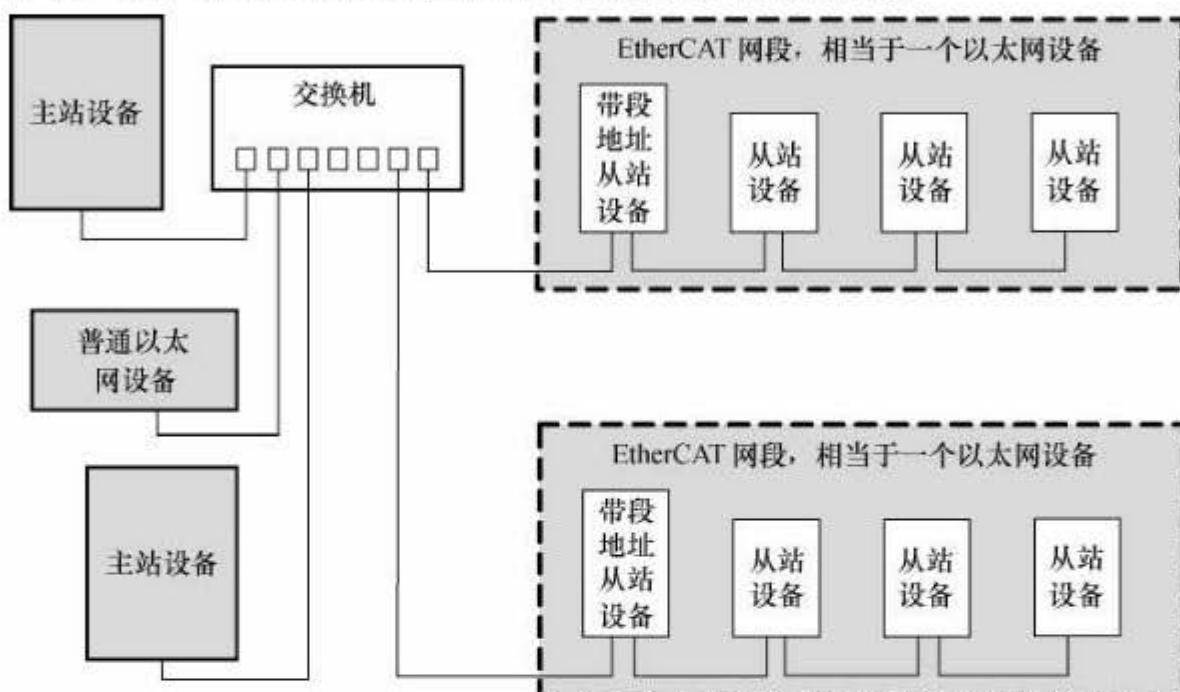


图 2.11 开放模式中的 EtherCAT 网段

6字节	6字节	2字节	2字节	44~1498字节	4字节
目的地址: 网段MAC地址	源地址: 主站MAC地址	帧类型 (0x88A4)	EtherCAT头	EtherCAT数据	FCS

图 2.12 开放模式下 EtherCAT 网段寻址地址内容

2.3.2 设备寻址

在设备寻址时，EtherCAT 子报文头内的 32 位地址分为 16 位从站设备地址和 16 位从站设备内部物理存储空间地址，如图 2.13 所示。16 位从站设备地址可以寻址 65 535 个从站设备，每个设备内最多可以有 64 K 字节的本地地址空间。



图 2.13 EtherCAT 设备寻址结构

设备寻址时，每个报文只寻址唯一的一个从站设备，但它有两种不同的设备寻址机制。

1. 顺序寻址

顺序寻址时，从站的地址由其在网段内的连接位置确定，用一个负数来表示每个从站在网段内由接线顺序决定的位置。顺序寻址子报文在经过每个从站设备时，其位置地址加1；从站在接收报文时，地址为0的报文就是寻址到自己的报文。由于这种机制在报文经过时更新设备地址，所以又被称为“自动增量寻址”。

图 2.14 中，网段中有三个从站设备，其顺序寻址的地址分别为 0、-1 和 -2。主站使用顺序寻址访问从站时子报文时的地址变化如图 2.15 所示。主站发出三个子报文分别寻址三个从站，其中的地址分别是 0、-1 和 -2，如图 2.15 中的数据帧 1。数据帧到达从站①时，从站①检查到子报文 1 中的地址为 0，从而得知子报文 1 就是寻址到自己的报文。数据帧经过从站①后，所有的顺序地址都增加 1，成为 1、0 和 -1，如图 2.15 中的数据帧 2。到达从站②时，从站②发现子报文 2 中的顺序地址为 0，即为寻址到自己的报文。同理，从站②也将所有子报文的顺序地址加 1，如图 2.15 中的数据帧 3。数据帧到达从站③时，子报文 3 中的顺序地址为 0，即为寻址从站③的报文。经过从站③处理后，数据帧成为图 2.15 中的数据帧 4。

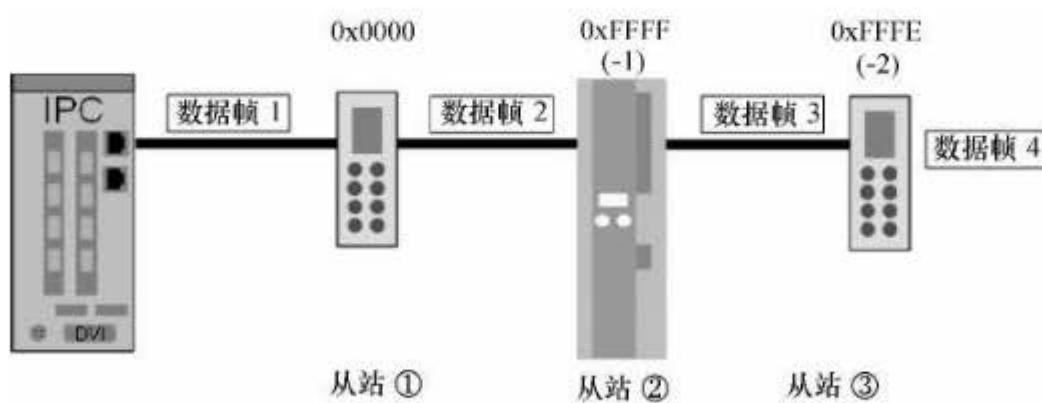


图 2.14 顺序寻址时的从站地址



图 2.15 顺序寻址时子报文地址的变化

在实际应用中, 顺序寻址主要用于启动阶段, 主站配置站点地址给各个从站。此后, 可以使用与物理位置无关的站点地址来寻址从站。这种寻址机制能自动为从站设定地址。

2. 设置寻址

设置寻址时, 从站的地址与其在网段内的连接顺序无关。如图 2.16 所示, 地址可以由主站在数据链路启动阶段配置给从站, 也可以由从站在上电初始化的时候从自身的配置数据存储区装载, 然后由主站在链路启动阶段使用顺序寻址方式读取各个从站的设置地址, 并在后续运行中使用。

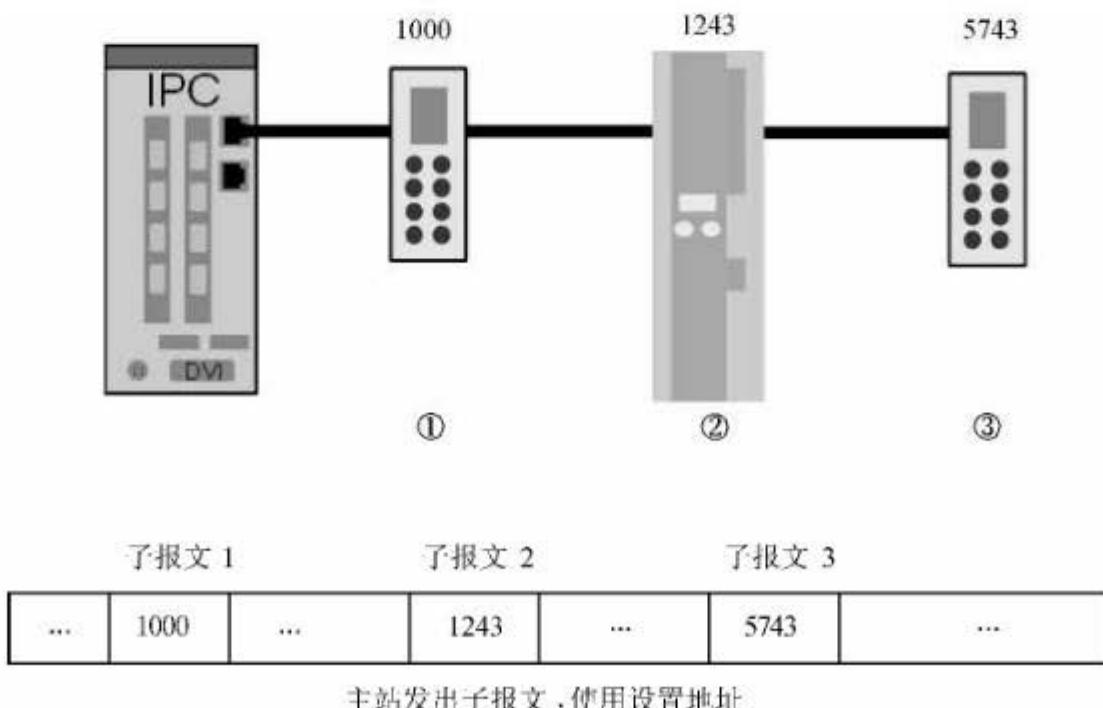


图 2.16 设置寻址时的从站地址和报文结构

2.3.3 逻辑寻址和FMMU

逻辑寻址时，从站地址并不是单独定义的，而是使用寻址段内 4 GB (2^{32}) 逻辑地址空间中的一段区域。报文内的 32 位地址区作为整体的数据逻辑地址完成设备的逻辑寻址。

逻辑寻址方式由现场总线内存管理单元 (FMMU, Fieldbus Memory Management Unit) 实现，FMMU 功能位于每一个 ESC 内部，将从站本地物理存储地址映射到网段内逻辑地址，其原理如图 2.17 所示。

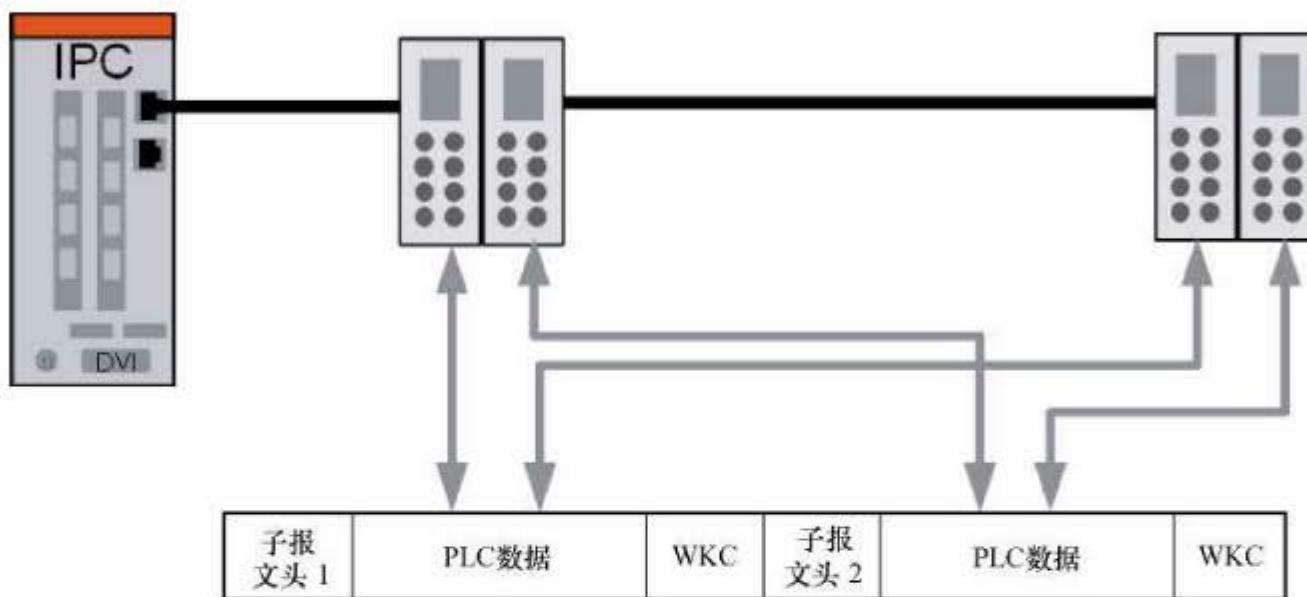


图 2.17 现场总线内存管理单元 (FMMU) 运行原理

FMMU 单元由主站设备配置，并在数据链路启动过程中传送给从站设备。每个 FMMU 单元需要以下配置信息：数据逻辑位起始地址、从站物理内存起始地址、位长度、表示映射方向（输入或输出）的类型位，从站设备内的所有数据都可以按位映射到主站逻辑地址。表 2.3 和图 2.18 是一个映射实例，将主站控制变量区 0x00014711 从第 3 位开始的 6 位数据映射到由设备地址 0x0F01 第 1 位开始的 6 位数据写操作。0x0F01 是一个开关量输出设备。

表 2.3 FMMU 配置示例

FMMU 配置寄存器	数 值
数据逻辑起始地址	0x00014711
数据长度（字节数，按跨字节计算）	2
数据逻辑起始位	3
数据逻辑终止位	0
从站物理内存起始地址	0x0F01
物理内存起始位	1
操作类型（1：只读，2：只写，3：读写）	2
激活（使能）	1

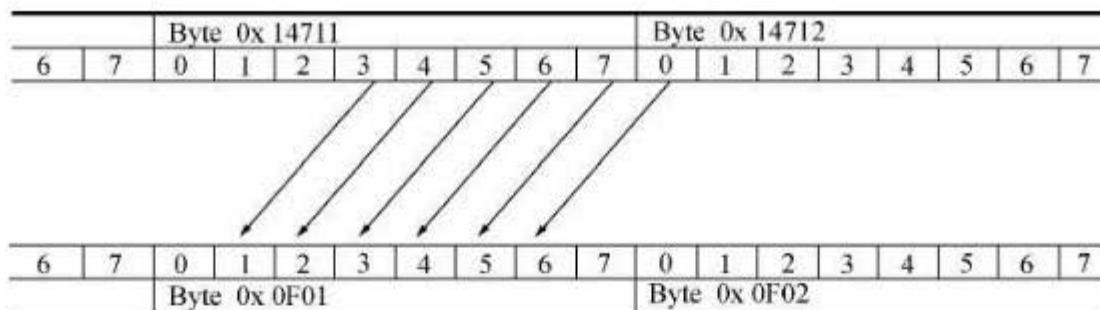


图 2.18 FMMU 映射举例

从站设备收到一个数据逻辑寻址的 EtherCAT 子报文时，检查是否有 FMMU 单元地址匹配。如果有，它将输入类型数据插入到 EtherCAT 子报文数据区的对应位置，以及从 EtherCAT 子报文数据区的对应位置抽取输出类型数据。使用逻辑寻址可以灵活地组织控制系统，优化系统结构。逻辑寻址方式特别适用于传输或交换周期性过程数据。FMMU 操作具有以下功能特点：

- 每个数据逻辑地址字节只允许被一个 FMMU 读和另一个 FMMU 写操作，或被同一个 FMMU 进行读写交换操作；
- 对一个逻辑地址的读写操作与使用一个 FMMU 读和另一个 FMMU 写操作具有相同的结果；
- 按位读写操作不影响报文中没有被映射到的其他位，因此允许将几个从站 ESC 中的位数据映射到主站的同一个逻辑字节；
- 读写一个未配置的逻辑地址空间不会改变其内容。

2.3.4 通信服务和 WKC

EtherCAT 子报文所有的服务都是以主站操作描述的。数据链路层规定了从站内部物理存储、读写和交换（读取并马上写入）数据的服务。读写操作和寻址方式共同决定了子报文的通信服务类型，由子报文头中的命令字节表示。EtherCAT 支持的所有命令如表 2.4 所列。

表 2.4 EtherCAT 通信服务命令

寻址方式	读写模式	命令名称和编号	解 释	WKC
空指令-		NOP (0)	没有操作	0
顺序寻址	读数据	APRD (1)	主站使用顺序寻址从从站读取一定长度数据	1
	写数据	APWR (2)	主站使用顺序寻址向从站写入一定长度数据	1
	读写	APRW (3)	主站使用顺序寻址与从站交换数据	3
设置寻址	读数据	FPRD (4)	主站使用设置寻址从从站读取一定长度数据	1
	写数据	FPWR (5)	主站使用设置寻址向从站写入一定长度数据	1
	读写	FPRW (6)	主站使用设置寻址与从站交换数据	3

续表 2.4

寻址方式	读写模式	命令名称和编号	解 释	WKC
广播寻址	读数据	BRD (7)	主站从所有从站的物理地址读取数据并做逻辑或操作	与寻址到从站个数相关
	写数据	BWR (8)	主站广播写入所有从站	
	读写	BRW (9)	主站与所有从站交换数据，对读取的数据做逻辑或操作	
逻辑寻址	读数据	LRD (10)	使用逻辑地址读取一定长度数据	
	写数据	LWR (11)	使用逻辑地址写入一定长度数据	
	读写	LRW (12)	使用逻辑寻址与从站交换数据	
顺序寻址	读，多重写	ARMW (13)	由从站读取数据，并写入以后所有从站的相同地址	
设置寻址		FRMW (14)		

主站接收到返回数据帧后，检查子报文中的 WKC，如果不等于预期值，则表示此子报文没有被正确处理。子报文的 WKC 预期值与通信服务类型和寻址地址相关。子报文经过某一个从站时，如果是单独地读或写操作，WKC 加 1。如果是读写操作，读成功时 WKC 加 1，写成功时 WKC 加 2，读写全部完成时 WKC 加 3。子报文由多个从站处理时，WKC 是各个从站处理结果的累加。

2.4 分布时钟

分布时钟 (DC, Distributed Clock) 可以使所有 EtherCAT 设备使用相同的系统时间，从而控制各设备任务的同步执行。从站设备可以根据同步的系统时间产生同步信号，用于中断控制或触发数字量输入输出。支持分布式时钟的从站称为 DC 从站。分布时钟具有以下主要功能：

- 实现从站之间时钟同步；
- 为主站提供同步时钟；
- 产生同步的输出信号；
- 为输入事件产生精确的时间标记；
- 产生同步的中断；
- 同步更新数字量输出；
- 同步采样数字量输入。

2.4.1 分布时钟描述

分布时钟机制使所有的从站都同步于一个参考时钟。主站连接的第一个具有分布时钟功能的从站作为参考时钟，以参考时钟来同步其他设备和主站的从时钟。为了实现精确的时钟同步控制，必须测量和计算数据传输延时和本地时钟偏移，并补偿本地时钟的漂移。

同步时钟所涉及到如下 6 个的时间概念。

(1) 系统时间

系统时间是分布时钟使用的系统计时。系统时间从 2000 年 1 月 1 日零点开始，使用 64 位二进制变量表示，单位为纳秒 (ns)，最大可以计时 500 y。也可以使用 32 位二进制变量表示，32 位时间值最大可以表示 4.2 s，通常用于通信和时间标记。

(2) 参考时钟和从时钟

EtherCAT 协议规定主站连接的第一个具有分布时钟功能的从站作为参考时钟，其他从站的时钟称为从时钟。参考时钟被用于同步其他从站设备的从时钟和主站时钟。参考时钟提供 EtherCAT 系统时间。

(3) 主站时钟

EtherCAT 主站也具有计时功能，称为主站时钟。主站时钟可以在分布时钟系统中作为从时钟被同步。在初始化阶段，主站可以按照系统时间的格式发送主站时间给参考时钟从站，使分布时钟使用系统时间计时。

(4) 本地时钟、其初始偏移量和时钟漂移

每一个 DC 从站都有本地时钟，本地时钟独立运行，使用本地时钟信号计时。系统启动时，各从站的本地时钟和参考时钟之间有一定的差值，称为时钟初始偏移量。在运行过程中，由于参考时钟和 DC 从站时钟使用各自的时钟源等原因，它们的计时周期存在一定的漂移，这将导致时钟运行不同步，本地时钟产生漂移。因此，必须对时钟初始偏移和时钟漂移都进行补偿。

(5) 本地系统时间

每个 DC 从站的本地时钟经过补偿和同步之后都产生一个本地系统时间，分布时钟同步机制就是使各个从站的本地系统时间保持一致。参考时钟也是相应从站的本地系统时钟。

(6) 传输延时

数据帧在从站之间传输时会产生一定的延迟，其中包括设备内部和物理连接延迟。所以在同步从时钟时，应该考虑参考时钟与各个从时钟之间的传输延时。

以上各时间量的定义如表 2.5 所列。

表 2.5 分布式时钟系统时间量定义

符 号	描 述
t_{sys_ref}	参考时钟时间，作为系统时间
$t_{local}(n)$	各从站本地时钟的时间，独立运行
$t_{sys_local}(n)$	各从站本地系统时间，同步后应该等于 t_{sys_ref}
$T_{offset}(n)$	从时钟和参考时钟之间的初始偏移量
$T_{delay}(n)$	参考时钟到各从时钟之间的传输延时

2.4.2 传输延时和时钟初始偏移量的测量

分布时钟初始化时，首先测量参考时钟到所有其他从时钟之间的传输延时，将其写入

各从站；并计算得到从时钟与参考时钟之间的偏移量，将其写入从时钟站。

测量原理如图 2.19 所示。横坐标为参考时钟 t_{sys_ref} ，纵坐标为某一个从时钟的本地时间 $t_{local}(n)$ ，假设 $t_{local}(n) > t_{sys_ref}$ ，它们的关系由下式确定：

$$t_{local}(n) = t_{sys_ref} + T_{offset}(n) \quad (2-1)$$

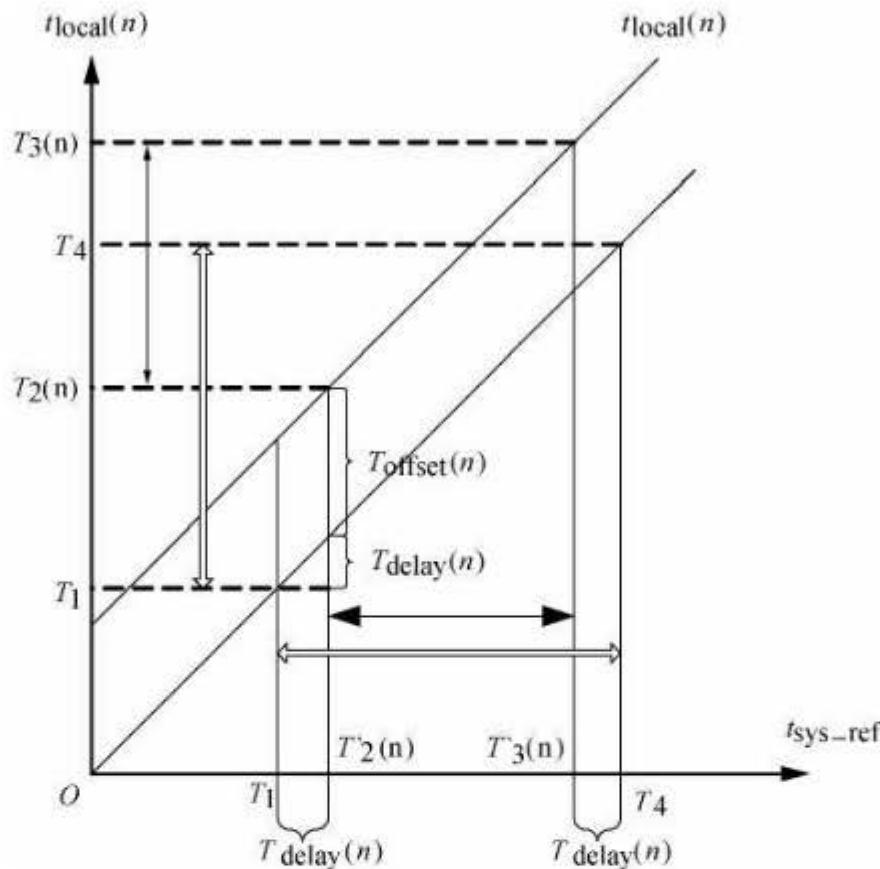


图 2.19 传输延时和时钟初始偏移量测量原理

传输延时和时钟初始偏移量的测量和计算步骤如下：

① 主站发送一个广播写命令数据帧，数据帧到达每个从站后每个从站设备分别保存每个端口接收到以太网帧前导符的第一位（SOF, Start Of Frame）的时刻。根据图 2.19，数据帧到达参考时钟从站时 t_{sys_ref} 为 T_1 时刻，到达从站 n 时从时钟本地时钟 $t_{local}(n)$ 的时刻为 $T_2(n)$ ，可以建立以下关系：

$$T_2(n) - T_1 = T_{offset}(n) + T_{delay}(n) \quad (2-2)$$

整理后成为：

$$T_{offset}(n) = T_2(n) - T_1 - T_{delay}(n) \quad (2-3)$$

② 数据帧经过所有的从站后返回时，到达从站 n 时本地时钟 $t_{local}(n)$ 的时刻为 $T_3(n)$ ，到达参考时钟从站 t_{sys_ref} 的时刻为 T_4 ；

③ 假设线缆延时均匀，并且所有从站设备的处理和转发延时都一样，根据图 2.19 中

的几何关系，从站 n 到参考时钟的传输延时可以由下式计算：

$$T_{\text{delay}}(n) = [(T_4 - T_1) - (T_3(n) - T_2(n))] / 2 = [(T_4 - T_1) - (T_3(n) - T_2(n))] / 2 \quad (2-4)$$

主站读取从站保存的时间值，使用公式 (2-4) 计算各个从站的传输延时 $T_{\text{delay}}(n)$ ，并写入到各个从站中；为了得到准确的传输延时，主站可以多次测量，然后求平均值；在初始化后的运行中也可以随时测量传输延时，以补偿环境变化对传输延时的影响。

④ 主站由公式 (2-3) 计算出初始偏移量 $T_{\text{offset}}(n)$ ，并写入各个从站。初始偏移量只用于对从时钟的粗略同步且只需要测量一次。

2.4.3 时钟同步

每个设备的本地时钟是自由运行的，会与参考时钟产生漂移。为了使所有设备都以相同的绝对系统时间运行，主站计算参考时钟与每个从站设备时钟之间的偏移量 $T_{\text{offset}}(n)$ ，并写入从站，以便计算从时钟的本地系统时间。利用 $T_{\text{offset}}(n)$ 可以在不改变自由运行的本地时钟的情况下实现时钟同步。每个 DC 从站使用自己的本地时间 $t_{\text{local}}(n)$ 和本地偏移量 $T_{\text{offset}}(n)$ 通过公式 (2-5) 计算它的本地系统时间副本。这个时间用来产生同步信号和锁存信号时间标记，供从站微处理器使用。

$$t_{\text{sys_local}}(n) = t_{\text{local}}(n) - T_{\text{offset}}(n) \quad (2-5)$$

在测得传输延时和时钟初始偏移量之后，主站开始同步各从站的时钟。主站使用 ARMW 或 FRMW 命令发送数据报文，从参考时钟从站读取它的当前系统时间 $t_{\text{sys_ref}}$ 并写入从时钟的从站设备中。每个从时钟从站的时间控制环在数据帧的 SOF 到达时锁存本地时钟时刻 $t_{\text{local}}(n)$ ，根据公式 (2-5) 计算得到本地系统时间 $t_{\text{sys_local}}(n)$ 。根据接收到的参考时钟系统时间 $t_{\text{sys_ref}}$ ，并利用本地保存的传输延时 $T_{\text{delay}}(n)$ ，计算得到本地时钟漂移量 Δt ：

$$\Delta t = t_{\text{sys_local}}(n) - T_{\text{delay}}(n) - t_{\text{sys_ref}} = t_{\text{local}}(n) - T_{\text{offset}}(n) - T_{\text{delay}}(n) - t_{\text{sys_ref}} \quad (2-6)$$

如果 Δt 是正数，表示本地时钟运行比参考时钟快，必须减慢运行。如果 Δt 是负数，表示本地时钟运行比参考时钟慢，必须加快运行。时间控制环路调整本地时钟的运行速度。正常情况下，ESC 控制本地时间每 10 ns 增加 10 个单位。当 $\Delta t > 0$ 时，则每 10 ns 增加 9，当 $\Delta t < 0$ 时，则增加 11，以实现时钟漂移补偿，如图 2.20 所示。

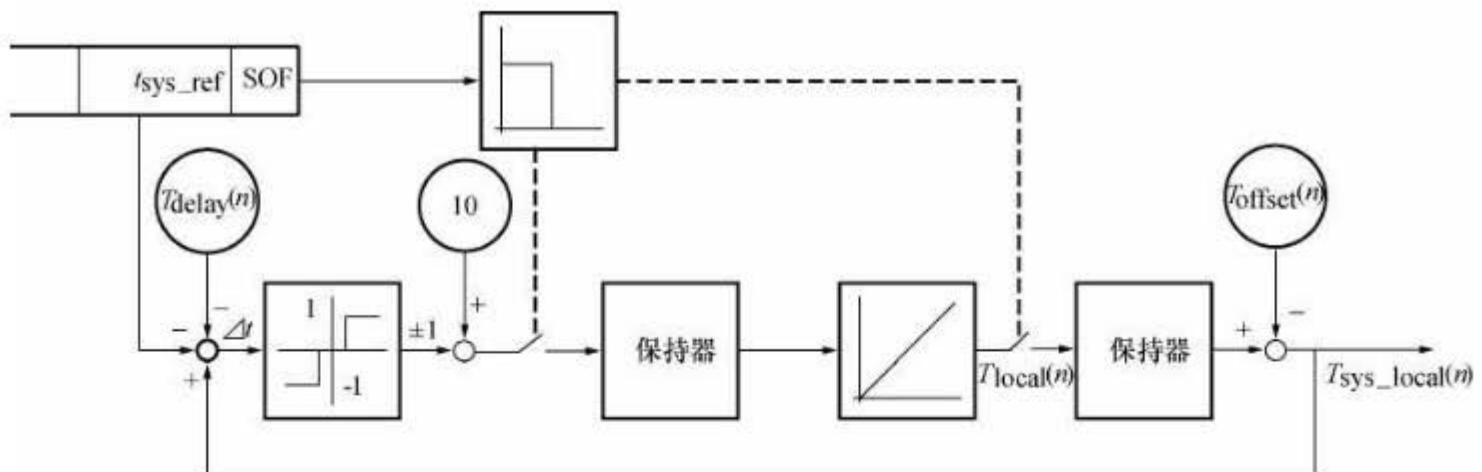


图 2.20 分布时钟同步原理

为了快速补偿时钟的初始偏差，主站应该在测量传输延时和偏移补偿之后在独立的数据帧中连续发送很多 ARMW/FRMW 命令，使从站时钟同步，完成分布式时钟初始化。随后，在周期性运行阶段，可以随着过程数据周期性地发送 ARMW/FRMW 命令读取参考时钟系统时间，写入到其他 DC 从站，实时补偿动态时钟漂移。发送时钟同步数据帧的周期时间必须保证从时钟的漂移小于控制应用所规定的限制。

根据对分布式时钟的支持情况，从站可以分为三种类型：

(1) 从站完全支持分布式时钟

具有接收时间标记和系统时间功能，根据应用要求产生同步信号或锁存信号时间标记。

(2) 从站只支持传输延时测量

3 个端口以上的从站设备必须支持传输延时的测量，需要本地时钟和锁存数据帧到达时刻功能。

(3) 从站不支持分布式时钟

只有两个端口的从站可以不支持分布式时钟。它们的处理和转发延时被周围的支持分布式时钟的从站作为物理连接延迟处理。

2.5 通信模式

在实际自动化控制系统中，应用程序之间通常有两种数据交换形式：时间关键 (time-critical) 和非时间关键 (non-time-critical)。时间关键表示特定的动作必须在确定的时间窗口内完成。如果不能在要求的时间窗口内完成通信，则有可能引起控制失效。时间关键的数据通常周期性发送，称为周期性过程数据通信。非时间关键数据可以非周期性发送，在 EtherCAT 中采用非周期性邮箱 (mailbox) 数据通信。

2.5.1 周期性过程数据通信

周期性过程数据通信通常使用 FMMU 进行逻辑寻址，主站可以使用逻辑读、写或读写命令同时操作多个从站。在周期性数据通信模式下，主站和从站有多种同步运行模式。

1. 从站设备同步运行模式

从站设备有以下三种同步运行模式。

(1) 自由运行

在自由运行模式下，本地控制周期由一个本地定时器中断产生。周期时间可以由主站设定，这是从站的可选功能。自由运行模式的本地周期如图 2.21 所示。其中 T_1 为本地微处理器从 ESC 复制数据并计算输出数据的时间； T_2 为输出硬件延时； T_3 为输入锁存偏移时间。这些参数反映了从站的时间性能。

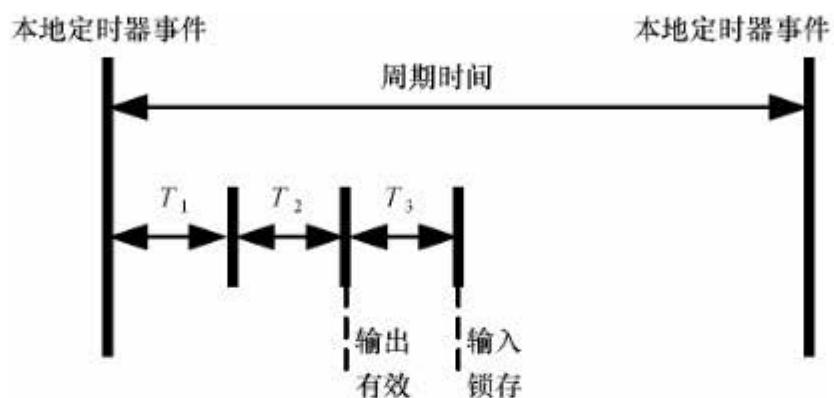


图 2.21 自由运行模式的本地周期

(2) 同步于数据输入或输出事件

本地周期在发生数据输入或输出事件的时候触发，如图 2.22 所示。主站可以将过程数据帧的发送周期写给从站，从站可以检查是否支持这个周期时间或对周期时间进行本地优化。从站可以选择支持这个功能。通常同步于数据输出事件，如果从站只有输入数据，则同步于数据输入事件。

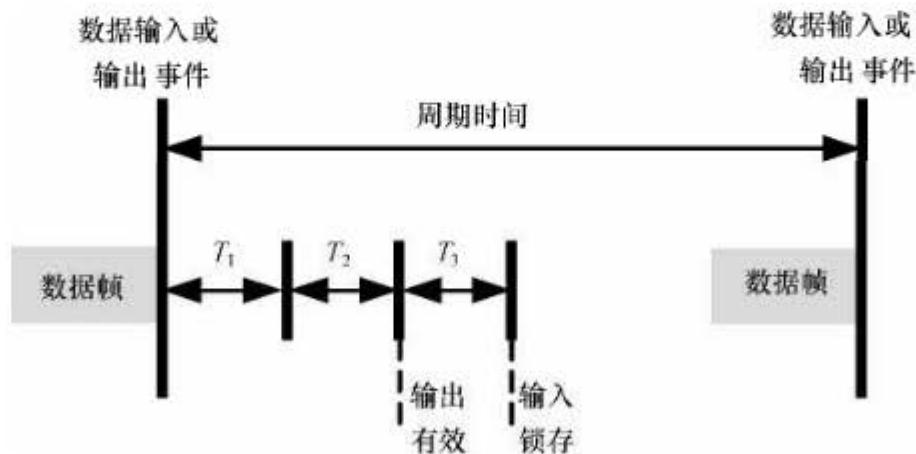


图 2.22 同步于数据输入或输出事件的本地周期

(3) 同步于分布式时钟同步事件

本地周期由 SYNC 事件触发，如图 2.23 所示。主站必须在 SYNC 事件之前完成数据帧的发送。此时要求主站时钟也要同步于参考时钟。

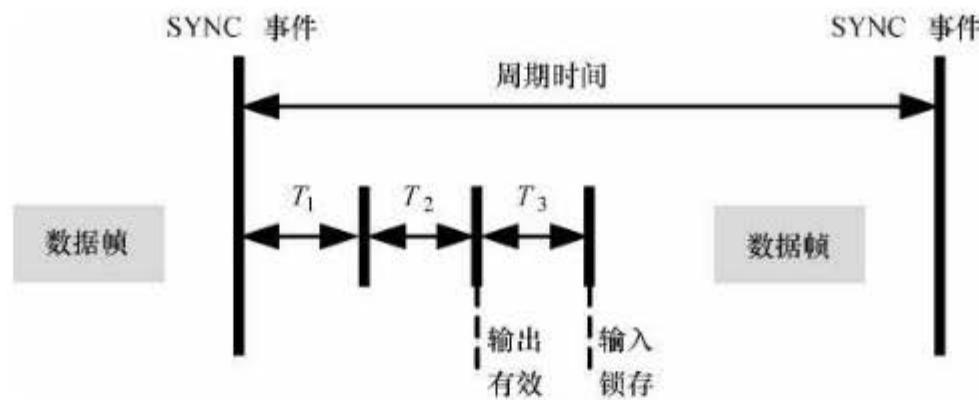


图 2.23 同步于 SYNC 事件的本地周期

为了进一步优化从站同步性能，从站应该在数据收发事件发生时从接收到的过程数据帧复制输出数据，然后等待 SYNC 信号到达后继续本地操作，如图 2.24 所示。数据帧必须比 SYNC 信号提前 T_1 时间到达，从站在 SYNC 事件之前已经完成数据交换和控制计算，接收 SYNC 信号后可以马上执行输出操作，从而进一步提高同步性能。

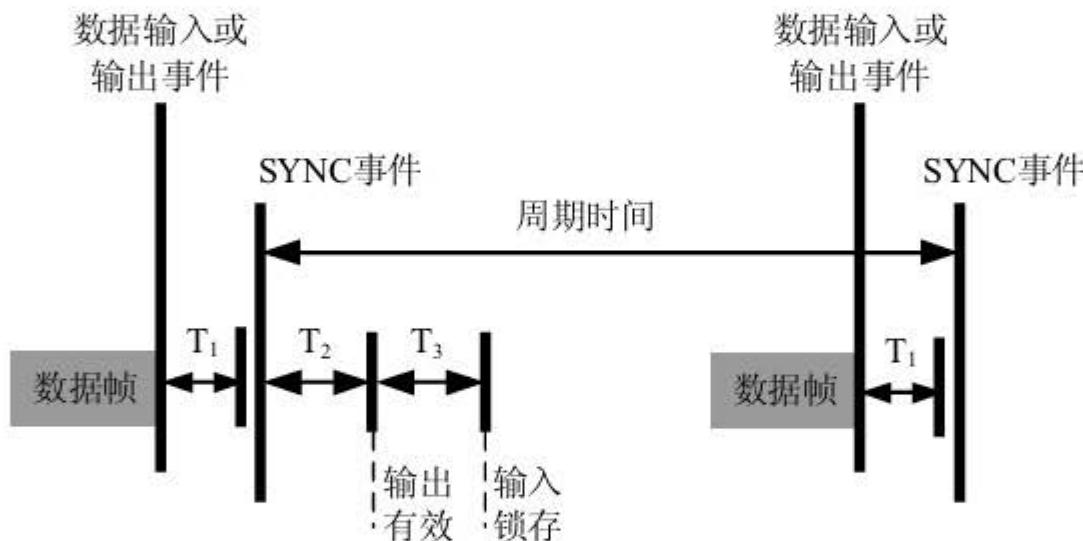


图 2.24 优化的同步于 SYNC 事件的本地周期

2. 主站设备同步运行模式

主站有以下两种同步模式：

(1) 周期性模式

在周期性模式下，主站周期性地发送过程数据帧。主站周期通常由一个本地定时器控制。从站可以运行在自由运行模式或同步于接收数据事件模式。对于运行在同步模式的从站，主站应该检查相应的过程数据帧的周期时间，保证大于从站支持的最小周期时间。

主站可以以不同的周期时间发送多种周期性的过程数据帧，以便获得最优化的带宽。例如，以较小的周期发送运动控制数据，以较大的周期发送 I/O 数据。

(2) DC 模式

在 DC 模式下，主站运行与周期性模式类似，只是主站本地周期应该和参考时钟同步。主站本地定时器应该根据发布参考时钟的 ARMW 报文进行调整。在运行过程中，用于动

态补偿时钟漂移的 ARMW 报文返回主站后，主站时钟可以根据读回的参考时钟时间进行调整，使之大致同步于参考时钟时间。

DC 模式下，所有支持 DC 的从站都应该同步于 DC 系统时间。主站也应该使其通信周期同步于 DC 参考时钟时间。图 2.25 表示主站本地周期与 DC 参考时钟同步的工作原理。

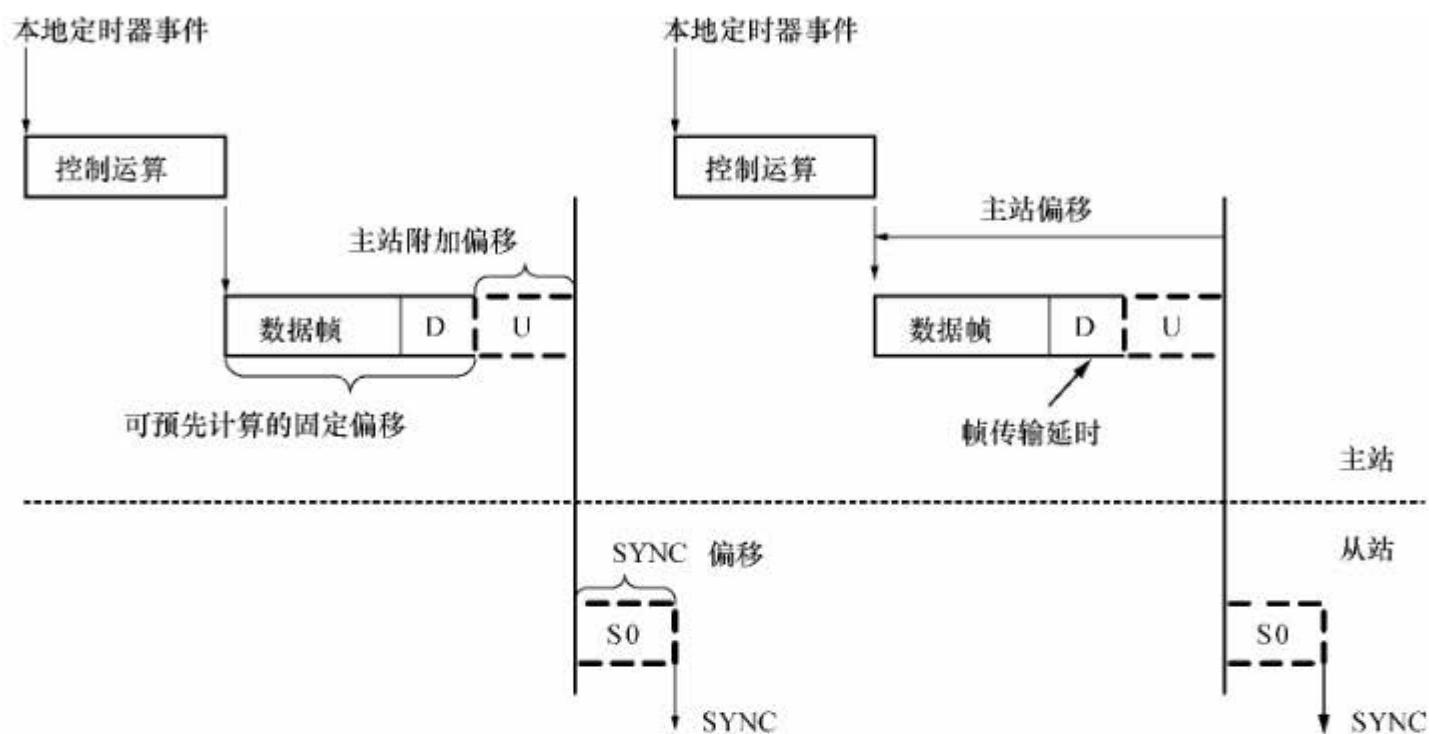


图 2.25 主站 DC 模式

主站本地运行由一个本地定时器启动。本地定时器应该比 DC 参考时钟定时存在一个提前量，提前量为以下时间之和：

- 控制程序执行时间；
- 数据帧传输时间；
- 数据帧传输延时 D；
- 附加偏移（与各从站延迟时间的抖动和控制程序执行时间的抖动值有关，用于主站周期时间的调整）。

2.5.2 非周期性邮箱数据通信

EtherCAT 协议中非周期性数据通信称为邮箱数据通信，它可以双向进行——主站到从站和从站到主站。它支持全双工、两个方向独立通信和多用户协议。从站到从站的通信由主站作为路由器来管理。邮箱通信数据头中包括一个地址域，使主站可以重寄邮箱数据。邮箱数据通信是实现参数交换的标准方式，如果需要配置周期性过程数据通信或需要其他非周期性服务时需要使用邮箱数据通信。

邮箱数据报文结构如图 2.26 所示。通常邮箱通信只对应一个从站，所以报文中使用设备寻址模式。其数据头中各数据元素的解释如表 2.6 所列。

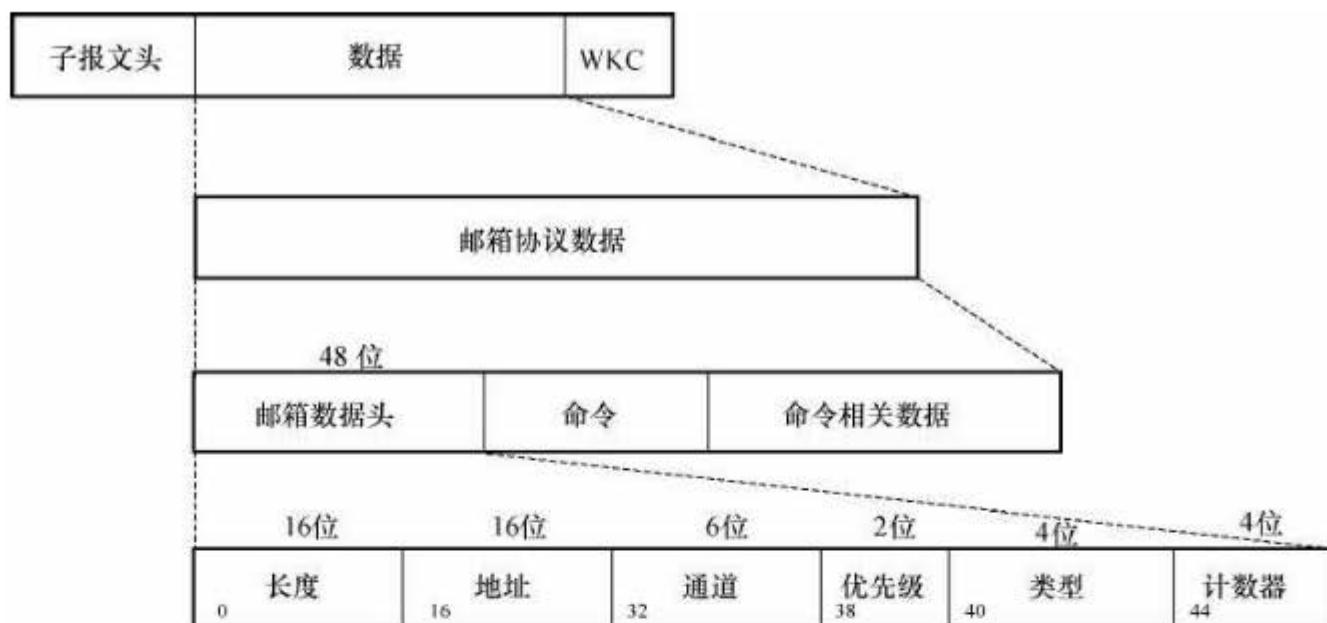


表 2.6 邮箱数据头

数据元素	位 数	描 述
长度	16 位	跟随的邮箱服务数据长度
地址	16 位	主站到从站通信时, 为数据源从站地址 从站到从站通信时, 为数据目的从站地址
通道	6 位	保留
优先级	2 位	保留
类型	4 位	邮箱类型, 后续数据的协议类型, 0: 邮箱通信出错 2: EoE (Ethernet over EtherCAT) 3: CoE (CANopen over EtherCAT) 4: FoE (File Access over EtherCAT) 5: SoE (Servo Drive over EtherCAT) 15: VoE (Vendor specific profile over EtherCAT)
计数器 Ctr	4 位	用于重复检测的顺序编号, 每个新的邮箱服务将加 1 (为了兼容老版本而只使用 1~7)

(1) 主站到从站通信——写邮箱命令

主站发送写数据区命令将发送邮箱数据给从站。主站需要检查从站邮箱命令应答报文中工作计数器 WKC。如果工作计数器为 1, 表示写命令成功。反之, 如果工作计数器没有增加, 通常因为从站没有读完上一个命令, 或在限定的时间内没有响应, 主站必须重发写邮箱数据命令。

(2) 从站到主站通信——读邮箱命令

从站有数据要发送给主站，必须先将数据写入输入邮箱缓存区，然后由主站来读取。主站发现从站 ESC 输入邮箱数据区有数据等待发送时，会尽快发送适当的读命令来读取从站数据。主站有两种方法来测定从站是否已经将邮箱数据填入输入数据区。一种是使用FMMU周期性地读某一个标志位。使用逻辑寻址可以同时读取多个从站的标志位，但其缺点是每个从站都需要一个FMMU单元。另一个方法是简单地轮询ESC输入邮箱的数据区。读命令的工作计数器增加1表示从站已经将新数据填入了输入数据区。

邮箱通信出错时，应答数据定义如表2.7所列。

表2.7 邮箱通信错误时应答数据定义列表

数据元素	长 度	描 述
命令	16位	0x01: 邮箱命令
命令相关数据	16位	0x01: 邮箱语法错误 0x02: 不支持邮箱协议 0x03: 邮箱通道无效 0x04: 不支持邮箱服务 0x05: 邮箱头无效 0x06: 邮箱数据太短 0x07: 邮箱服务内存不足 0x08: 邮箱数据数目错误

2.6 状态机和通信初始化

EtherCAT状态机(ESM, EtherCAT State Machine)负责协调主站和从站应用程序在初始化和运行时的状态关系。

EtherCAT设备必须支持四种状态，另外还有一个可选的状态。

- ① Init: 初始化，简写为I;
- ② Pre-Operational: 预运行，简写为P;
- ③ Safe-Operational: 安全运行，简写为S;
- ④ Operational: 运行，简写为O;
- ⑤ Boot-Strap: 引导状态(可选)，简写为B。

以上各状态之间的转化关系如图2.27所示。从初始化状态向运行状态转化时，必须按照“初始化→预运行→安全运行→运行”的顺序转化，不可以越级转化。从运行状态返回时可以越级转化。引导状态为可选状态，只允许与初始化状态之间互相转化。所有的状态改变都由主站发起，主站向从站发送状态控制命令请求新的状态，从站响应此命令，执行所请求的状态转换，并将结果写入从站状态指示变量。如果请求的状态转换失败，从站将给出错误标志。

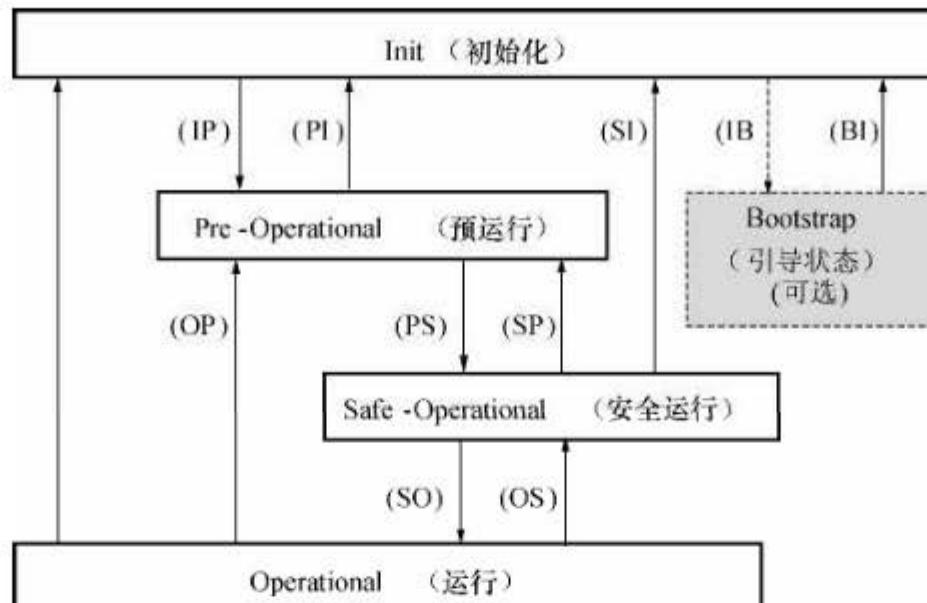


图 2.27 EtherCAT 状态转化关系

(1) Init: 初始化

初始化状态定义了主站与从站在应用层的初始通信关系。此时，主站与从站应用层不可以直接通信，主站使用初始化状态来初始化 ESC 的一些配置寄存器。如果从站支持邮箱通信，则配置邮箱通道参数。

(2) Pre-Operational: 预运行

在预运行状态下，邮箱通信被激活。主站与从站可以使用邮箱通信来交换与应用程序相关的初始化操作和参数。在这个状态下不允许过程数据通信。

(3) Safe-Operational: 安全运行

在安全运行状态下，从站应用程序读入输入数据，但是不产生输出信号。设备无输出，处于“安全状态”。此时，仍然可以使用邮箱通信。

(4) Operational: 运行

在运行状态下，从站应用程序读入输入数据，主站应用程序发出输出数据，从站设备产生输出信号。此时，仍然可以使用邮箱通信。

(5) Boot-Strap: 引导状态（可选）

引导状态的功能是下载设备固件程序。主站可以使用 FoE 协议的邮箱通信下载一个新的固件程序给从站。

表 2.8 总结了 EtherCAT 状态转化操作和初始化过程。

表 2.8 EtherCAT 状态及其转化过程总结

状态和状态转化	操作
初始化	应用层没有通信，主站只能读写 ESC 寄存器
初始化向预运行转化 Init to Pre-Op(IP)	主站配置从站站点地址寄存器 如果支持邮箱通信，则配置邮箱通道参数 如果支持分布式时钟，则配置 DC 相关寄存器 主站写状态控制寄存器，以请求“Pre-Op”状态

续表 2.8

状态和状态转化	操作
预运行	应用层邮箱数据通信
Pre-Op to Safe-Op (PS)	主站使用邮箱初始化过程数据映射 主站配置过程数据通信使用的 SM 通道 主站配置 FMMU 主站写状态控制寄存器，以请求“Safe-Op”状态
安全运行	应用层支持邮箱数据通信： 有过程数据通信，但是只允许读输入数据，不产生输出信号
Safe-Op to Op(SO)	主站发送有效的输出数据 主站写状态控制寄存器，以请求“Op”状态
运行状态	输入和输出全部有效 仍然可以使用邮箱通信

2.7 应用层协议

应用层 AL (Application Layer) 是 EtherCAT 协议最高的一个功能层，是直接面向控制任务的一层，它为控制程序访问网络环境提供手段，同时为控制程序提供服务。应用层不包括控制程序，它只是定义了控制程序与网络交互的接口，使符合此应用层协议的各种应用程序可以协同工作，如图 2.28 所示。EtherCAT 包括以下几种应用层协议：

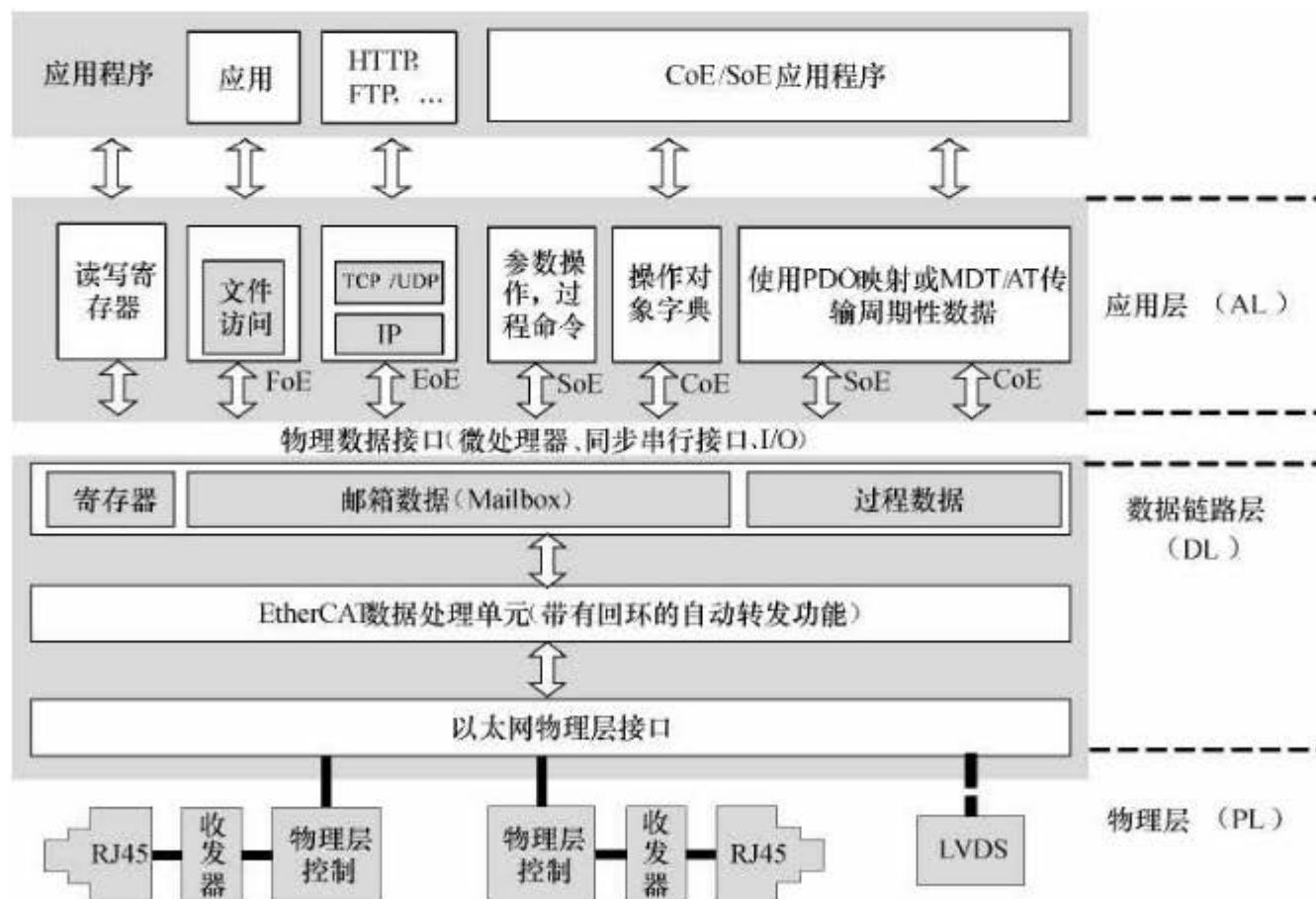


图 2.28 EtherCAT 协议结构

(1) CANopen over EtherCAT (CoE)

CANopen 最初是为 CAN (Control Area Network) 总线控制系统所开发的应用层协议。EtherCAT 协议在应用层支持 CANopen 协议，并作了相应的扩充。主要功能有：

- 使用邮箱通信访问 CANopen 对象字典及其对象，实现网络初始化；
- 使用 CANopen 应急对象和可选的事件驱动 PDO 消息，实现网络管理；
- 使用对象字典映射过程数据，周期性传输指令数据和状态数据。

(2) Servo Drive over EtherCAT (SoE)

IEC61491 是国际上第一个专门用于伺服驱动器控制的实时数据通信协议标准，其商业名称为 SERCOS (Serial Real-time Communication Specification)。EtherCAT 协议的通信性能非常适合数字伺服驱动器的控制，应用层使用 SERCOS 应用层协议实现数据接口，可以实现以下功能：

- 使用邮箱通信访问伺服控制规范参数 (IDN)，配置伺服系统参数；
- 使用 SERCOS 数据电报格式配置 EtherCAT 过程数据报文，周期性传输伺服指令数据和伺服状态数据。

(3) Ethernet over EtherCAT (EoE)

除了前面描述的主从站设备之间的通信寻址模式外，EtherCAT 也支持 IP 标准的协议，比如 TCP/IP、UDP/IP 和所有其他高层协议 (HTTP 和 FTP 等)。EtherCAT 能分段传输标准以太网协议数据帧，并在相关的设备完成组装。这种办法可以避免为长数据帧预留时间片，大大缩短周期性数据的通信周期。此时，主站和从站需要相应的 EoE 驱动程序支持。

(4) File Access over EtherCAT (FoE)

该协议通过 EtherCAT 下载和上传固件程序和其他文件，其使用类似 TFTP (Trivial File Transfer Protocol, 简单文件传输协议) 的协议，不需要 TCP/IP 的支持，实现简单。

第3章 EtherCAT 从站控制芯片

EtherCAT 从站控制芯片 ESC 是实现 EtherCAT 数据链路层协议的专用集成电路芯片。它处理 EtherCAT 数据帧，并为从站控制装置提供数据接口。ESC 结构如图 3.1 所示，这具有以下主要功能：

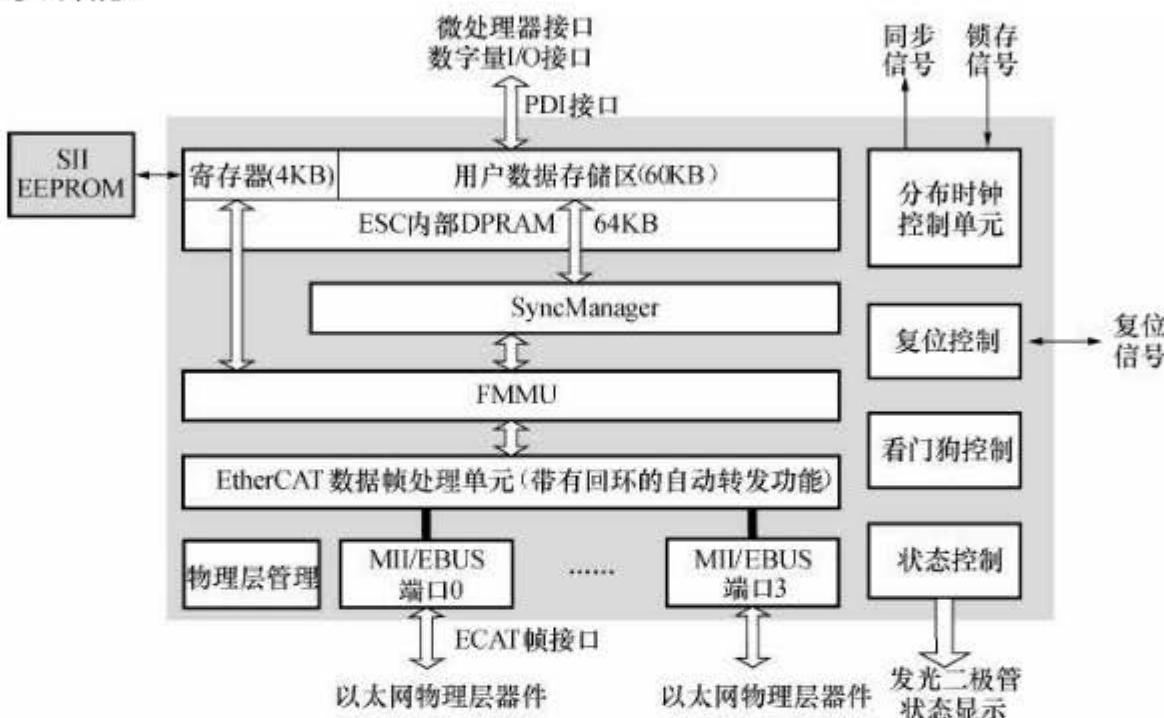


图 3.1 ESC 结构图

- 集成数据帧转发处理单元，通信性能不受从站微处理器性能限制。每个 ESC 最多可以提供 4 个数据收发端口；主站发送 EtherCAT 数据帧操作被 ESC 称为 ECAT 帧操作；
- 最大 64 K 字节的双端口存储器 DPRAM 存储空间，其中包括 4K 字节的寄存器空间和 1~60 K 字节的用户数据区，DPRAM 可以由外部微处理器使用并行或串行数据总线访问，访问 DPRAM 的接口称为物理设备接口 PDI (Physical Device Interface)；
- 可以不用微处理器控制，作为数字量输入/输出芯片独立运行，具有通信状态机处理功能，最多提供 32 位数字量输入输出；
- 具有 FMMU 逻辑地址映射功能，提高数据帧利用率；
- 由储存同步管理器通道 SyncManager (SM) 管理 DPRAM，保证了应用数据的一致性和安全性；
- 集成分布时钟 (Distribute Clock) 功能，为微处理器提供高精度的中断信号；
- 具有 EEPROM 访问功能，存储 ESC 和应用配置参数，定义从站信息接口 (SII, Slave Information Interface)。

3.1 ESC 概述

3.1.1 ESC 芯片种类

ESC 由德国 BECKHOFF 自动化有限公司提供，包括 ASIC 芯片和 IP-Core。目前有 2 种规格的 ASIC 从站控制专用芯片：ET1100 和 ET1200，如表 3.1 所列。

表 3.1 EtherCAT 通信 ASIC 芯片

特性	ET1100	ET1200
端口数	4 个端口，使用 EBUS 或 MII 模式	3 个端口，最多 1 个 MII 端口
FMMU 单元	8	3
存储同步管理单元	8	4
过程数据 RAM	8K 字节	1K 字节
分布式时钟	64 位	32 位
物理设备接口 PDI	32 位数字量 IO 8/16 位异步/同步微处理器接口 MCI (Micro Controller Interface) 串行接口 SPI	16 位数字量 IO SPI (Series Periphery Interface)
EEPROM 容量	16kbit	16kbit — 4Mbit
封装	BGA128, 10×10 mm	QFN48, 7×7 mm

用户也可以使用 IP-Core 将 EtherCAT 通信功能集成到设备控制 FPGA 中，并根据需要配置功能和规模。使用 Altra 公司 Cyclone 系列 FPGA 的 IP-Core 的 ET18xx 功能如表 3.2 所列。

表 3.2 IP-Core 功能配置

特 性	FPGA 的 IP-Core 的 ET18xx
端口数	2 个 MII 或 RMII (Reduced MII) 端口
FMMU 单元	0~8 个可配置
存储同步管理单元	0~8 个可配置
过程数据 RAM	1~60 K 字节可配置
分布式时钟	可配置
物理设备接口	32 位数字量 IO 8/16 位异步/同步微处理器接口 SPI (Series Periphery Interface) Avalon/OPB 片上总线

3.1.2 ESC 存储空间

ESC 芯片具有 64K 字节的 DPRAM 地址空间，前 4K（0x0000~0x0FFF）字节的空间为寄存器空间。0x1000~0xFFFF 的地址空间为过程数据存储空间，不同的芯片类型所包含的过程数据空间有所不同，如图 3.2 所示。

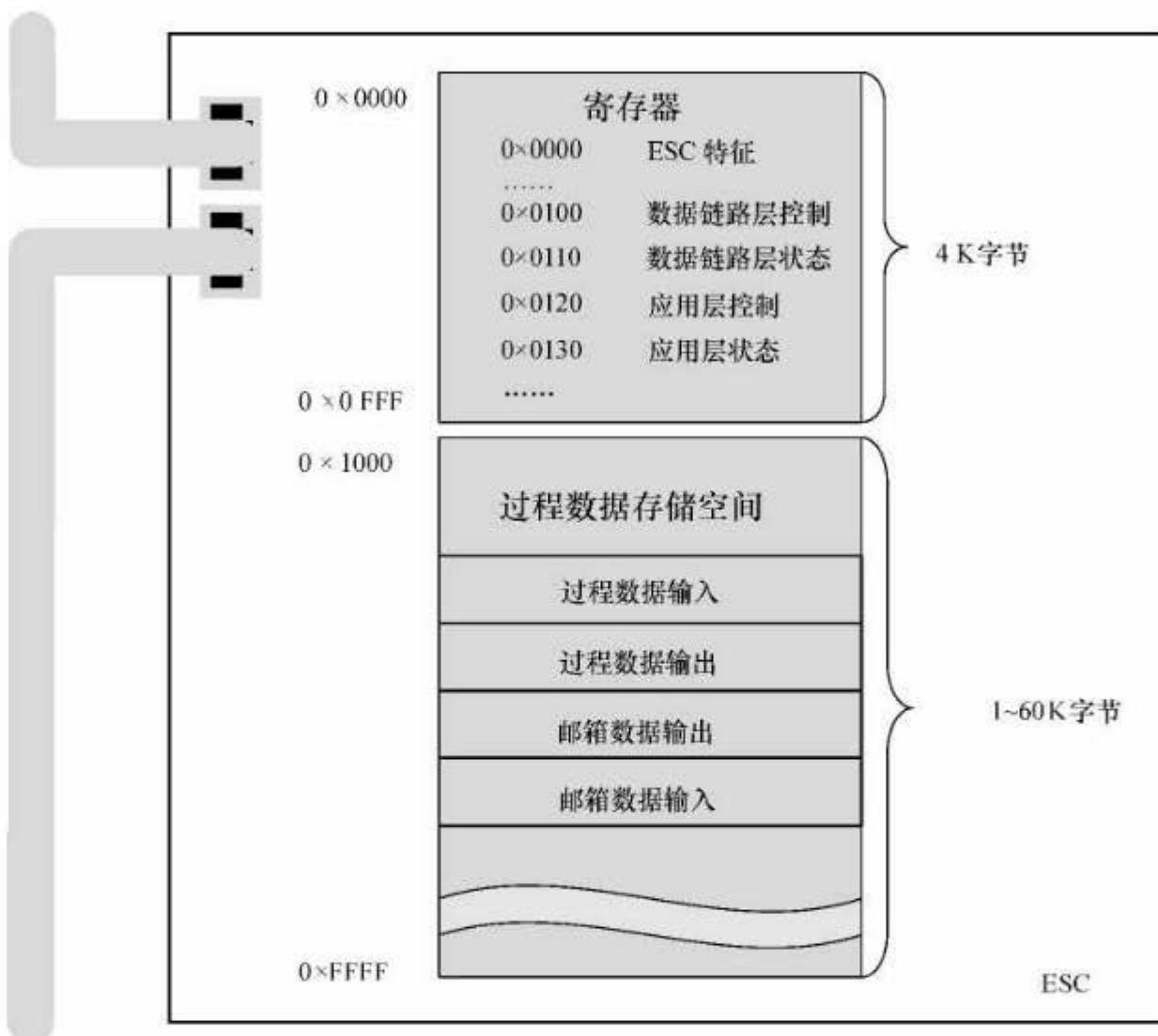


图 3.2 ESC 内部存储空间示意图

0×0000~0×0F7F 的寄存器具有缓存区，ESC 在接收到一个写寄存器操作数据帧时，数据首先存放在缓存区中。如果确认数据帧接收正确，缓存区中的数值将被传送到真正的寄存器中，否则不接收缓存区中的数据。也就是说，寄存器内容在正确接收到 EtherCAT 数据帧的 FCS 之后才被刷新。用户和过程数据存储区没有缓存区，所以对它的写操作将立即生效。如果数据帧接收错误，ESC 将不向上层应用控制程序通知存储区数据的改变。ESC 存储空间的详细定义如表 3.3 所列。

表3.3 ET1100存储空间定义

地址	长度 (字节)	描述	读/写	
			ECAT 帧	PDI
ESC信息				
0x0000	1	类型	R/-	R/-
0x0001	1	版本号	R/-	R/-
0x0002: 0x0003	2	内部标号	R/-	R/-
0x0004	1	FMMU数	R/-	R/-
0x0005	1	SM数	R/-	R/-
0x0006	1	RAM大小	R/-	R/-
0x0007	1	端口描述	R/-	R/-
0x0008: 0x0009	2	特性	R/-	R/-
站点地址				
0x0010: 0x0010	2	配置站点地址	R/W	R/-
0x0012: 0x0013	2	配置站点别名	R/-	R/W
写保护				
0x0020	1	寄存器写使能	-/W	-/-
0x0021	1	寄存器写保护	R/W	R/-
0x0030	1	写使能	-/W	-/-
0x0031	1	写保护	R/W	R/-
ESC复位				
0x0040	1	复位控制	R/W	R/-
数据链路层				
0x0100: 0x0103	4	数据链路控制	R/W	R/-
0x0108: 0x0109	2	物理读/写偏移	R/W	R/-
0x0110: 0x0111	2	数据链路状态	R/-	R/-
应用层				
0x0120: 0x0121	2	应用层控制	R/W	R/-
0x0130: 0x0131	2	应用层状态	R/-	R/W
0x0134: 0x0135	2	应用层状态码	R/-	R/W
物理设备接口 PDI (Physical Device Interface)				
0x0140: 0x0141	2	PDI控制	R/-	R/-
0x0150	1	PDI配置	R/-	R/-
0x0151	1	SYNC/LATCH 接口配置	R/-	R/-
0x0152: 0x0153	2	扩展PDI配置	R/-	R/-
中断				

续表 3.3

地址	长 度 (字节)	描 述	读/写	
0x0200: 0x0201	2	ECAT 中断屏蔽	R/W	R/-
0x0204: 0x0207	4	应用层中断事件屏蔽	R/-	R/W
0x0210: 0x0211	2	ECAT 中断请求	R/-	R/-
0x0220: 0x0223	4	应用层中断事件请求	R/-	R/-
错误计数器				
0x0300: 0x0307	4×2	接收错误计数器	R/W (clr)	R/-
0x0308: 0x030B	4	转发接收错误计数器	R/W (clr)	R/-
0x030C	1	ECAT 处理单元错误计数器	R/W (clr)	R/-
0x030D	1	PDI 错误计数器	R/W (clr)	R/-
0x0310: 0x0313	4	链接丢失计数器	R/W (clr)	R/-
看门狗设置				
0x0400: 0x0401	2	看门狗分频器	R/W	R/-
0x0410: 0x0411	2	PDI 看门狗计时器	R/W	R/-
0x0420: 0x0421	2	过程数据看门狗计时器	R/W	R/-
0x0440: 0x0441	2	过程数据看门狗状态	R/-	R/-
0x0442	1	过程数据看门狗超时计数器	R/W (clr)	R/-
0x0443	1	PDI 看门狗超时计数器	R/W (clr)	R/-
EEPROM 控制接口				
0x0500	1	EEPROM 配置	R/W	R/-
0x0501	1	EEPROM PDI 访问状态	R/-	R/W
0x0502: 0x0503	2	EEPROM 控制/状态	R/W	R/W
0x0504: 0x0507	4	EEPROM 地址	R/W	R/W
0x0508: 0x050F	8	EEPROM 数据	R/W	R/W
MII 管理接口				
0x0510: 0x0511	2	MII 管理控制/状态	R/W	R/W
0x0512	1	PHY 地址	R/W	R/W
0x0513	1	PHY 寄存器地址	R/W	R/W
0x0514: 0x0515	2	PHY 数据	R/W	R/W
0x0516	1	MII 管理 ECAT 操作状态	R/W	R/-
0x0517	1	MII 管理 PDI 操作状态	R/W	R/W
0x0518: 0x051B	4	PHY 端口状态	R/-	R/-
FMMU 配置寄存器				
0x0600: 0x06FF	16x16	FMMU[15:0]		
+0x0:0x3	4	逻辑起始地址	R/W	R/-

续表 3.3

地址	长 度 (字节)	描 述	读/写	
+0x4:0x5	2	长度	R/W	R/-
+0x6	1	逻辑起始位	R/W	R/-
+0x7	1	逻辑停止位	R/W	R/-
+0x8:0x9	2	物理起始地址	R/W	R/-
+0xA	1	物理起始位	R/W	R/-
+0xB	1	FMMU 类型	R/W	R/-
+0xC	1	FMMU 激活	R/W	R/-
+0xD:0xF	3	保留	R/-	R/-
存储同步管理器配置寄存器				
0x0800: 0x087F	16x16	同步管理器 SM[15:0]		
+0x0:0x1	2	物理起始地址	R/W	R/-
+0x2:0x3	2	长度	R/W	R/-
+0x4	1	SM 控制寄存器	R/W	R/-
+0x5	1	SM 状态寄存器	R/-	R/-
+0x6	1	激活	R/W	R/-
+0x7	1	PDI 控制	R/-	R/W
分布时钟控制寄存器				
0x0900: 0x09FF		分布时钟 DC 控制		
DC-接收时间				
0x0900: 0x0903	4	端口 0 接收时间	R/W	R/-
0x0904: 0x0907	4	端口 1 接收时间	R/-	R/-
0x0908: 0x090B	4	端口 2 接收时间	R/-	R/-
0x090C: 0x090F	4	端口 3 接收时间	R/-	R/-
DC-时钟控制环单元				
0x0910: 0x0917	4/8	系统时间	R/W	R/W
0x0918: 0x091F	4/8	数据帧处理单元接收时间	R/-	R/-
0x0920: 0x0927	4/8	系统时间偏移	R/W	R/W
0x0928: 0x092B	4	系统时间延迟	R/W	R/W
0x092C: 0x092F	4	系统时间漂移	R/-	R/-
0x0930: 0x0931	2		R/W	R/W
0x0932: 0x0933	2		R/-	R/-
0x0934	1	系统时间偏移过滤深度	R/W	R/W
0x0935	1		R/W	R/W
DC-周期性单元控制				

0x0980	1	周期单元控制	R/W	R/-
DC-SYNC 输出单元				
0x0981	1	激活	R/W	R/W
0x0982:0x0983	2	SYNC 信号脉冲宽度	R/-	R/-
0x098E	1	SYNC0 信号状态	R/-	R/-
0x098F	1	SYNC1 信号状态	R/-	R/-
0x0990:0x0997	4/8	周期性运行开始时间/下个 SYNC0 脉冲时间	R/W	R/W
0x0998:0x099F	4/8	下个 SYNC1 脉冲时间	R/-	R/-
0x09A0:0x09A3	4	SYNC0 周期时间	R/W	R/W
0x09A4:0x09A7	4	SYNC1 周期时间	R/W	R/W
DC-锁存单元				
0x09A8	1	Latch0 控制	R/W	R/W
0x09A9	1	Latch1 控制	R/W	R/W
0x09AE	1	Latch0 状态	R/-	R/-
0x09AF	1	Latch1 状态	R/-	R/-
0x09B0:0x09B7	4/8	Latch0 上升沿时间	R/-	R/-
0x09B8:0x09BF	4/8	Latch0 下降沿时间	R/-	R/-
0x09C0:0x09C7	4/8	Latch1 上升沿时间	R/-	R/-
0x09C8:0x09CF	4/8	Latch1 下降沿时间	R/-	R/-
DC-SM 时间				
0x09F0:0x09F3	4	EtherCAT 缓存改变事件时间	R/-	R/-
0x09F8:0x09FB	4	PDI 缓存开始事件时间	R/-	R/-
0x09FC:0x09FF	4	PDI 缓存改变事件时间	R/-	R/-
ESC 特征寄存器				
0x0E00:0x0EFF	256	ESC 特征寄存器, 例如: 上电值, 产品和厂商 ID		
数字量输入和输出				
0xF00:0xF03	4	数字量 I/O 输出数据	R/W	R/-
0xF10:0xF17	1-8	通用功能输出数据	R/W	R/W
0xF18:0xF1F	1-8	通用功能输入数据	R/-	R/-
用户 RAM/扩展特性				
0xF80:0xFFFF	128	用户 RAM/扩展 ESC 特性	R/W	R/W
过程数据 RAM				
0x1000:0x1003	4	数字量 I/O 输入数据	R/W	R/W
0x1000:0xFFFF	8K	过程数据 RAM	R/W	R/W

3.1.3 ESC 特征信息

ESC 芯片寄存器空间的前 10 个字节表示其基本配置性能，如表 3.4 所列。可以读取这些寄存器的值获取从站 ESC 芯片类型和功能。

表 3.4 ESC 特征寄存器

地 址	位	名 称	描 述	复位值
0x0000	0~7	类型	芯片类型 ET1100: 0x11 ET1200: 0x12	
0x0001	0~7	修订号	芯片版本修订号 IP Core: 主版本号 X	与 ESC 相关
0x0002~ 0x0003	0~15	内部版本号	内部版本号 IP Core: [7:4] = 子版本号 Y [3:0] = 维护版本号 Z	与 ESC 相关
0x0004	0~7	FMMU 支持	FMMU 通道数目 IP Core: 可配置 ET1100: 8 ET1200: 3	
0x0005	0~7	SM 支持	SM 通道数目 IP Core: 可配置 ET1100: 8 ET1200: 4	
0x0006	0~7	RAM 数量	过程数据存储区容量, 以 KByte 为单位 IP Core: 可配置 ET1100: 8 ET1200: 1	
0x0007	0~7	端口配置	4 个物理端口的用途 ESC 相关	
	1:0	Port 0	00: 没有实现	
	3:2	Port 1	01: 没有配置	
	5:4	Port 2	10: EBUS	
	7:6	Port 3	11: MII	
0x0008~ 0x0009	0	FMMU 操作	0: 按位映射 1: 按字节映射	0
	1	保留		
	2	分布时钟	0: 不支持 1: 支持 IP Core: 可配置 ET1100: 1 ET1200: 1	
	3	时钟容量	0: 32 位 1: 64 位 ET1100: 1 ET1200: 1 其他: 0	

续表 3.4

地 址	位	名 称	描 述	复位值
0x0008~0x0009	4	低抖动 EBUS	0: 不支持, 标准 EBUS 1: 支持, 抖动最小化	ET1100: 1 ET1200: 1 其他: 0
	5	增强的 EBUS 链接检测	0: 不支持 1: 支持, 如果在过去的 256 位中发现超过 16 个错误, 则关闭链接	ET1100: 1 ET1200: 1 其他: 0
	6	增强的 MII 链接检测	0: 不支持 1: 支持, 如果在过去的 256 位中发现超过 16 个错误, 则关闭链接	ET1100: 1 ET1200: 1 其他: 0
	7	分别处理 FCS 错误	0: 不支持 1: 支持	ET1100: 1 ET1200: 1 其他: 0
	8~15	保留		

3.2 ESC 芯片 ET1100

ET1100 是一种 EtherCAT 从站控制器 ESC 专用芯片。它具有 4 个数据收发端口、8 个 FMMU 单元、8 个 SM 通道、4K 字节控制寄存器、8K 字节过程数据储存器、支持 64 位的分布时钟功能。它可以直接作为 32 位数字量输入/输出站点, 或由外部微处理器控制, 组成复杂的从站设备。图 3.3 为 ET1100 的结构图。

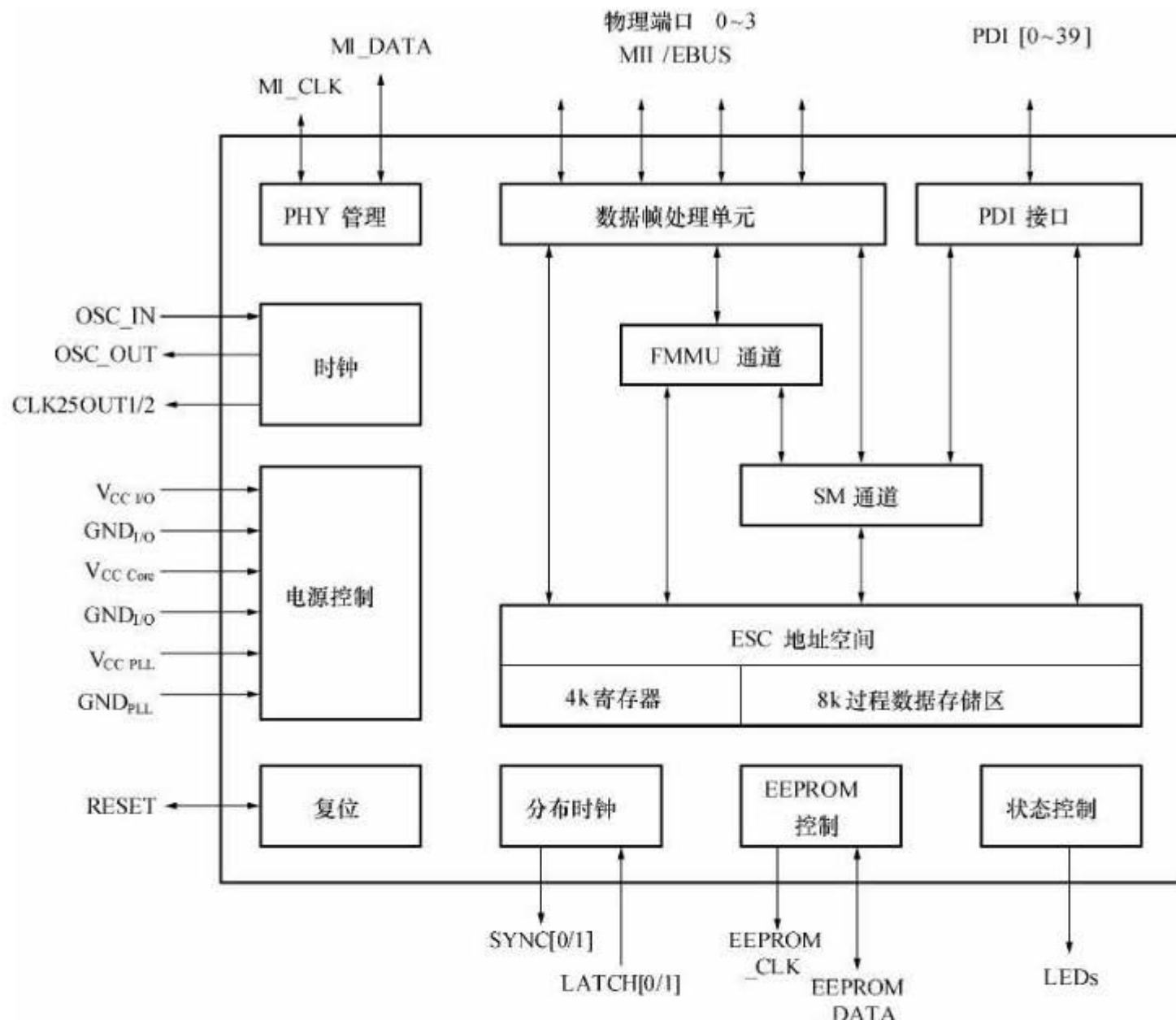


图 3.3 ET1100 结构框图

3. 2. 1 ET1100 引脚定义

ET1100 外形如图 3.4 所示，采用 BGA128 封装，引脚分布如图 3.5 所示，共有 128 个引脚。表 3.5 列出了 ET1100 的所有功能引脚，按照功能复用分类，包括 PDI 接口引脚、ECAT 帧接口引脚、芯片配置引脚和其他功能引脚。表 3.6 列出了其电源引脚。



图 3.4 ET1100 实物图

A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12
B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11	B12
C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12
D1	D2	D3	D4	D5	D6	D7	D8	D9	D10	D11	D12
E1	E2	E3	E4					E9	E10	E11	E12
F1	F2	F3	F4					F9	F10	F11	F12
G1	G2	G3	G4					G9	G10	G11	G12
H1	H2	H3	H4					H9	H10	H11	H12
J1	J2	J3	J4	J5	J6	J7	J8	J9	J10	J11	J12
K1	K2	K3	K4	K5	K6	K7	K8	K9	K10	K11	K12
L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L11	L12
M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11	M12

图 3.5 ET1100 芯片 BGA128 封装的引脚分布¹

表 3.5 ET1100 BGA128 封装信号引脚定义

功能 编号	PDI 接口				ECAT 帧接口		配置功能	其他功能
	PDI 编号	I/O 接口	MCI 接口	SPI 接口	MII 接口	EBUS 接口		
D12	PDI[0]	I/O[0]	*CS	SPI_CLK				
D11	PDI[1]	I/O[1]	*RD (*TS)	SPI_SEL				
C12	PDI[2]	I/O[2]	*WR (RD/*WR)	SPI_DI				
C11	PDI[3]	I/O[3]	*BUSY (*TA)	SPI_DO				
B12	PDI[4]	I/O[4]	*IRQ	SPI_IRQ				
C10	PDI[5]	I/O[5]	*BHE					
A12	PDI[6]	I/O[6]	EEPROM _Loaded	EEPROM _Loaded				
B11	PDI[7]	I/O[7]	ADR[15]					CPU_CLK
A11	PDI[8]	I/O[8]	ADR[14]	GPO[0]				SOF*
B10	PDI[9]	I/O[9]	ADR[13]	GPO[1]				OE_EXT
A10	PDI[10]	I/O[10]	ADR[12]	GPO[2]				OUTVALID
C9	PDI[11]	I/O[11]	ADR[11]	GPO[3]				WD_TRIG
A9	PDI[12]	I/O[12]	ADR[10]	GPI[0]				LATCH_IN

¹ 本表斜体部分表示可以通过配置引脚 CTRL_STATUS_MOVE 分配 PDI[23:16] 或 PDI [15:8] 作为控制/状态信号，其具体使用可参考本节“/数字量 I/O 接口”。

续表 3.5

功能 编号	PDI 接口				ECAT 帧接口		配置功能	其他功能
B9	PDI[13]	I/O[13]	ADR[9]	GPI[1]				<i>OE_CONF</i>
A8	PDI[14]	I/O[14]	ADR[8]	GPI[2]				<i>EEPROM_Loaded</i>
B8	PDI[15]	I/O[15]	ADR[7]	GPI[3]				
A7	PDI[16]	I/O[16]	ADR[6]	GPO[4]	RX_ERR(3)			<i>SOF</i>
B7	PDI[17]	I/O[17]	ADR[5]	GPO[5]	RX_CLK(3)			<i>OE_EXT</i>
A6	PDI[18]	I/O[18]	ADR[4]	GPO[6]	RX_D(3)[0]			<i>OUTVALID</i>
B6	PDI[19]	I/O[19]	ADR[3]	GPO[7]	RX_D(3)[2]			<i>WD_TRIG</i>
A5	PDI[20]	I/O[20]	ADR[2]	GPI[4]	RX_D(3)[3]			<i>LATCH_IN</i>
B5	PDI[21]	I/O[21]	ADR[1]	GPI[5]	LINK_MII(3)			<i>OE_CONF</i>
A4	PDI[22]	I/O[22]	ADR[0]	GPI[6]	TX_D(3)[3]			<i>EEPROM_Loaded</i>
B4	PDI[23]	I/O[23]	DATA[0]	GPI[7]	TX_D(3)[2]			
A3	PDI[24]	I/O[24]	DATA[1]	GPO[8]	TX_D(3)[1]	EBUS(3)-TX-		
B3	PDI[25]	I/O[25]	DATA[2]	GPO[9]	TX_D(3)[0]			
A2	PDI[26]	I/O[26]	DATA[3]	GPO[10]	TX_ENA(3)	EBUS(3)-TX+		
A1	PDI[27]	I/O[27]	DATA[4]	GPO[11]	RX_DV(3)	EBUS(3)-RX-		
B2	PDI[28]	I/O[28]	DATA[5]	GPI[8]	Err(3)/Trans(3)	Err(3)	RESET_VED	
B1	PDI[29]	I/O[29]	DATA[6]	GPI[9]	RX_D(3)[1]	EBUS(3)-RX+		
C2	PDI[30]	I/O[30]	DATA[7]	GPI[10]	LinkAct(3)		<i>P_CONF[3]</i>	
C1	PDI[31]	I/O[31]		GPI[11]	CLK25OUT2			
D1	PDI[32]	<i>SOF</i>	DATA[8]	GPO[12]	TX_D(2)[3]			
D2	PDI[33]	<i>OE_EXT</i>	DATA[9]	GPO[13]	TX_D(2)[2]			
E2	PDI[34]	<i>OUTVALID</i>	DATA[10]	GPO[14]	TX_D(2)[0]		<i>CTRL_STAT</i>	<i>US_MOVE</i>
G1	PDI[35]	<i>WD_TRIG</i>	DATA[11]	GPO[15]	RX_ERR(2)			
G2	PDI[36]	<i>LATCH_IN</i>	DATA[12]	GPI[12]	RX_CLK(2)			
H2	PDI[37]	<i>OE_CONF</i>	DATA[13]	GPI[13]	RX_D(2)[0]			
J2	PDI[38]	<i>EEPROM_Loaded</i>	DATA[14]	GPI[14]	RX_D(2)[2]			
K1	PDI[39]		DATA[15]	GPI[15]	RX_D(2)[3]			
F1					TX_ENA(2)	EBUS(2)-TX+		
E1					TX_D(2)[1]	EBUS(2)-TX-		
H1					RX_DV(2)	EBUS(2)-RX+		

续表 3.5

功能 编号	PDI 接口				ECAT 帧接口		配置功能	其他功能
J1					RX_D(2)[1]	EBUS(2)-RX-		
C3					Err(2)/Trans(2)	Err(2)	PHYAD_OFF	
E3					LinkAct(2)		P_CONF[2]	
F2					LINK_MII(2)	CLK25OUT1		CLK25OUT 1
M3					TX_ENA(1)	EBUS(1)-TX+		
L3					TX_D(1)[0]		TRANS_M ODE_ENA	
M2					TX_D(1)[1]	EBUS(1)-TX-		
L2					TX_D(1)[2]		P_MODE[0]	
M1					TX_D(1)[3]		P_MODE[1]	
L4					RX_D(1)[0]			
M5					RX_D(1)[1]	EBUS(1)-RX+		
L5					RX_D(1)[2]			
M6					RX_D(1)[3]			
M4					RX_DV(1)	EBUS(1)-RX-		
L6					RX_ERR(1)			
K4					RX_CLK(1)			
K3					LINK_MII(1)			
K2					Err(1)/Trans(1)	Err(1)	CLK_MODE[1]	
L1					LinkAct(1)	LinkAct(1)	P_CONF[1]	
M9					TX_ENA(0)	EBUS(0)-TX+		
L8					TX_D(0)[0]		C25_ENA	
M8					TX_D(0)[1]	EBUS(0)-TX-		
L7					TX_D(0)[2]		C25_SHI[0]	
M7					TX_D(0)[3]		C25_SHI[1]	
K10					RX_D(0)[0]			
M12					RX_D(0)[1]	EBUS(0)-RX+		
L11					RX_D(0)[2]			
L12					RX_D(0)[3]			
M11					RX_DV(0)	EBUS(0)-RX-		
M10					RX_ERR(0)			
L10					RX_CLK(0)			
L9					LINK_MII(0)			

续表 3.5

功能 编号	PDI 接口				ECAT 帧接口		配置功能	其他功能
J11					Err(0)/Trans(0)	Err(0)	CLK_MODE[0]	
J12					LinkAct(0)	LinkAct(0)	P_CONF[0]	
H11							EEPROM_SIZE	RUN
G12								OSC_IN
F12								OSC_OUT
H12								RESET
C4								RBIAS
H3								TESTMODE
G11								EEPROM_C_LK
F11								EEROM_D_ATA
K11							LINKPOL	MI_CLK
K12								MI_DATA
E11								SYNC/Latch[0]
E12								SYNC/Latch[1]

表 3.6 ET1100 供电引脚定义

引脚编号	电源功能
C5, D3, J3, K5, K8, J10, F10, D10, E9, F3, H9	V _{CCIO}
D5, D5, D4, J4, J5, J8, J9, F9, D9, H4, K9	GND _{IO}
C6, K6, K7, C7	V _{CC Core}
D6, J6, J7, D7	GND _{Core}
G10	V _{CC PLL}
G9	GND _{PLL}
E4, G3, G4, E10, C8, H10, F4, D8	Res.

3.2.2 物理通信端口

ET1100 有 4 个物理通信端口，分别命名为端口 0 到端口 3，每个端口都可以配置为 MII 接口或 EBUS 接口两种形式。

1.MII 接口

ET1100 使用 MII 接口时, 需要外接以太网物理层 PHY 芯片。为了降低处理/转发延时, ET1100 的 MII 接口省略了发送 FIFO。因此, ET1100 对以太网物理层芯片有一些附加的功能要求。ET1100 选配的以太网 PHY 芯片应该满足以下基本功能和附加要求。

(1) 基本功能

- 遵从 IEEE 802.3 100BaseTX 或 100BaseFX 规范;
- 支持 100 Mbit/s 全双工链接;
- 提供一个 MII 接口;
- 使用自动协商;
- 支持 MII 管理接口;
- 支持 MDI/MDI-X 自动交叉。

(2) 附加要求

- PHY 芯片和 ET1100 使用同一个时钟源;
- ET1100 不使用 MII 接口检测或配置连接, PHY 芯片必须提供一个信号指示是否建立了 100 Mbit/s 的全双工连接;
- PHY 芯片的连接丢失响应时间应小于 15 μ s, 以满足 EtherCAT 的冗余性能要求;
- PHY 的 TX_CLK 信号和 PHY 的输入时钟之间的相位关系必须固定, 最大允许 5 ns 的抖动;
- ET1100 不使用 PHY 的 TX_CLK 信号, 以省略 ET1100 内部的发送 FIFO;
- TX_CLK 和 TX_ENA 及 TX_D[3:0]之间的相移由 ET1100 通过设置 TX 相位偏移来补偿, 可以使 TX_ENA 及 TX_D[3:0]延迟 0、10、20 或 30 ns。

以上要求中, 时钟源最为重要。ET1100 的时钟信号包括 OSC_IN 和 OSC_OUT。时钟源的布局对系统设计的电磁兼容性能有很大的影响。需要满足以下条件:

- 时钟源尽可能靠近 ESC 布置;
- 这个区域的地层应该无缝;
- 电源对时钟源和 ESC 时钟低阻抗;
- 应该使用时钟元器件推荐的电容值;
- 时钟源和 ESC 时钟输入之间的电容量应该相同, 具体数值取决于线路板几何特性;
- ET1100 的时钟精度为 25 ppm 以上。

使用石英晶体时, OSC_IN 和 OSC_OUT 连接到 25 MHz 外部晶体的两端, ET1100 的 CLK25OUT1/2 输出作为 PHY 芯片的时钟输入, 如图 3.6 所示。图中电容的典型值为 12 pF。如果 ET1100 使用振荡器作为输入, PHY 芯片不能使用 CLK25OUT1/2 输出作为时钟源, 必须与 ET1100 使用同一振荡器作为输入, 如图 3.7 所示。

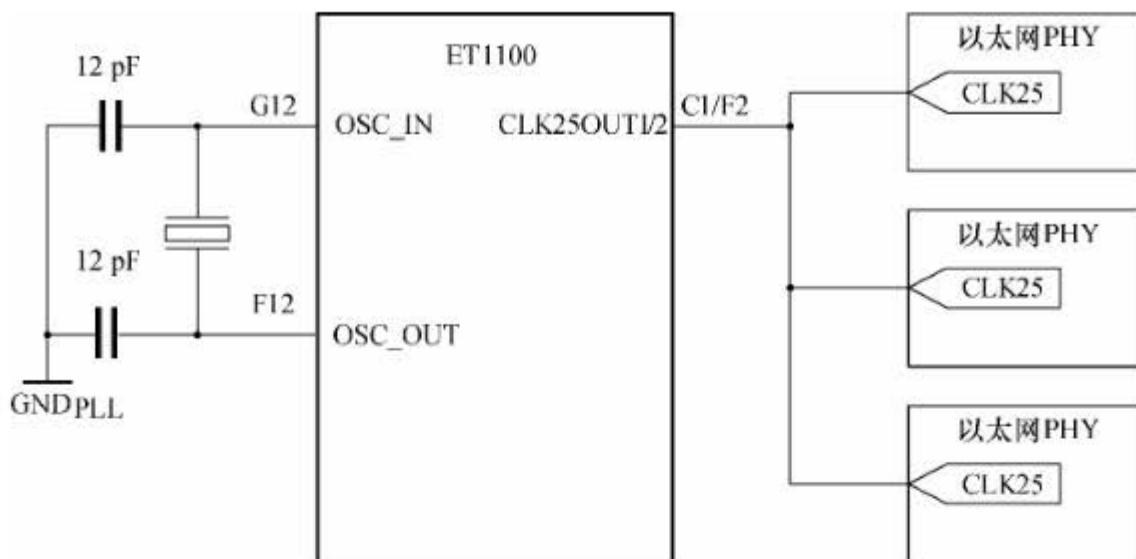


图 3.3 ET1100 使用石英晶体作为时钟源时的连接

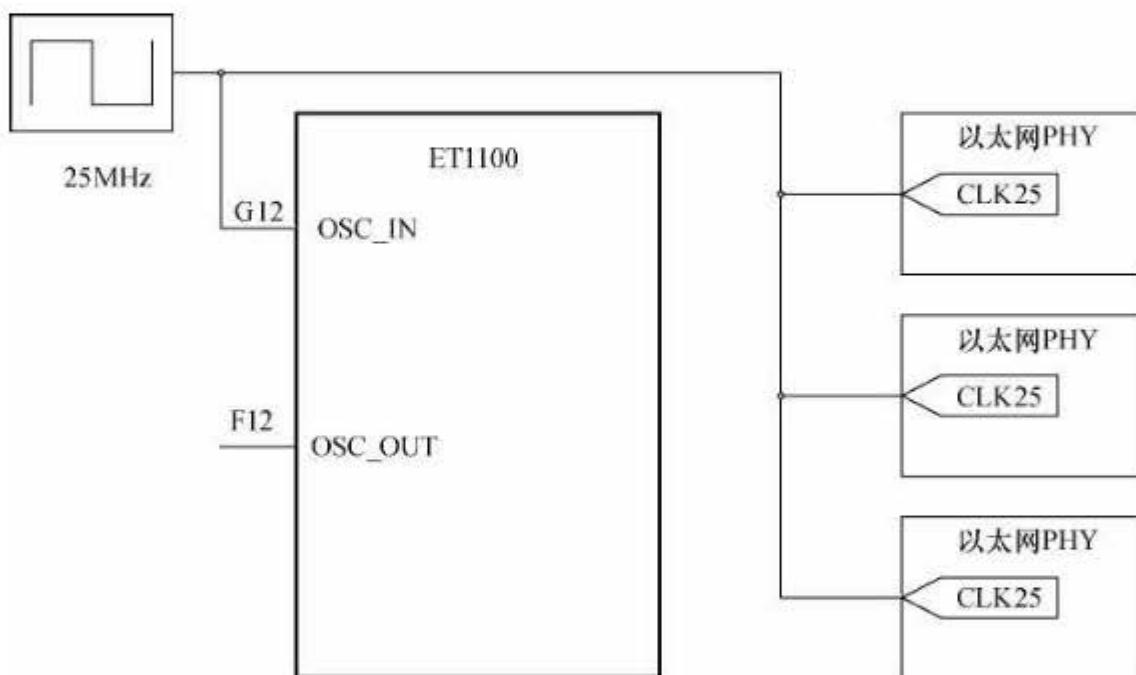


图 3.4 ET1100 使用振荡器作为时钟源输入时的连接

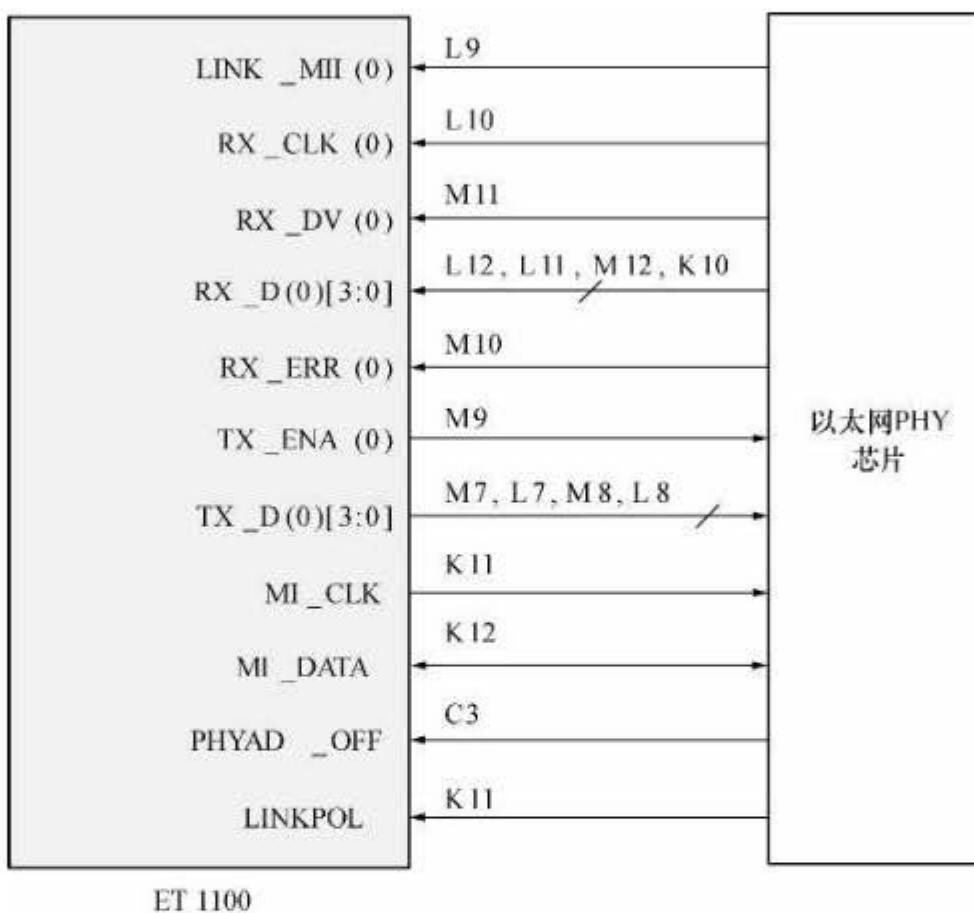
ET1100 没有使用标准 MII 接口的全部引脚信号，表 3.7 描述了 ET1100 所使用的 MII 接口信号，图 3.8 为端口为 0 的 MII 与 PHY 芯片连接示意图。

表 3.7 MII 接口信号描述

信号	方向	描述
LINK_MII	IN	100 Mbit/s 的全双工连接状态指示
RX_CLK	IN	接收时钟
RX_DV	IN	接收数据有效
RX_D[3:0]	IN	接收数据 (RXD)
RX_ERR	IN	接收出错 (RX_ER)

续表 3.7

TX_ENA	OUT	发送使能 (TX_EN)
TX_D[3:0]	OUT	发送数据 (TXD)
MI_CLK	OUT	管理接口时钟 (MDC)
MI_DATA	BIDIR	管理接口数据 (MDIO)
PHYAD_OFF	IN	PHY 地址偏移配置, 属于配置引脚
LINKPOL	IN	LINK_MII 极性配置, 属于配置引脚



ET 1100

图 3.5 端口为 0 的 MII 接口示意图

MI_DATA 应该连接外部上拉电阻, 推荐阻值为 $4.7\text{ k}\Omega$ 。MI_CLK 为轨到轨 (rail-to-rail) 驱动, 空闲时为高。

每个端口的 PHY 地址等于其逻辑端口号加 1 (PHYAD_OFF=0, PHY 地址为 1~4), 或逻辑端口号加 17 (PHYAD_OFF=1, PHY 地址为 17~20)。

2.EBUS/LVDS 接口

EtherCAT 协议自定义了一种物理层传输方式 EBUS。EBUS 传输介质使用低压差分信号 LVDS (Low Voltage Differential Signaling), 由 ANSI/TIA/EIA-644 “低压差分信号接口电路电气特性” 标准定义, 最远传输距离为 10 m。

图 3.9 为两个 ET1100 芯片使用 EBUS 连接的示意图。使用两对 LVDS 线对, 一对接收数据帧, 一对发送数据帧。每对 LVDS 线对只需要跨接一个 $100\ \Omega$ 的负载电阻, 不需要其他物理层元器件, 缩短了从站之间的传输延时, 减少了元器件。表 3.8 描述了 EBUS 接

口的各个信号。

EBUS 可以满足快速以太网 100 Mbits/s 的数据波特率。它只是简单地封装以太网数据帧，所以可以传输任意以太网数据帧，而不只是 EtherCAT 帧。

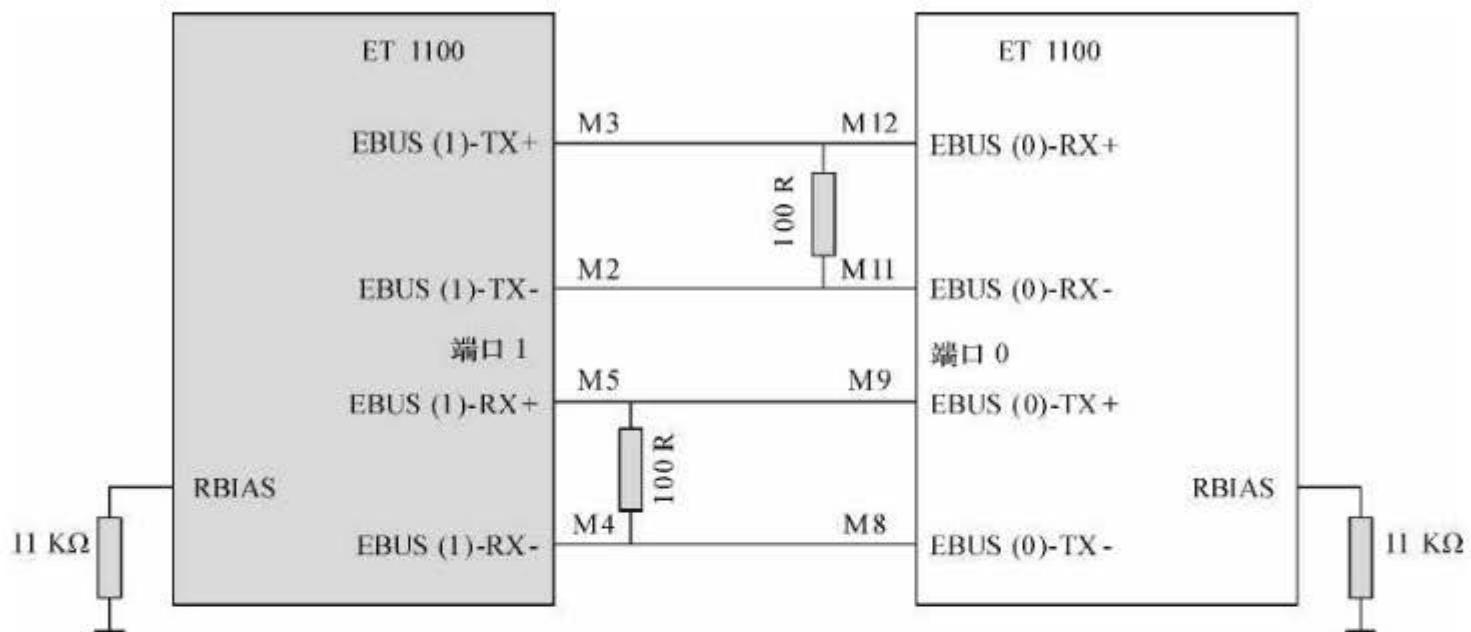


图 3.6 EBUS 接口信号

表 3.1 EBUS 信号描述

信 号	方 向	描 述
EBUS-TX+	输出	EBUS/LVDS 发送信号
EBUS-TX-		
EBUS-RX+	输入	EBUS/LVDS 接收信号
EBUS-RX-		
RBIAS		用于 EBUS-TX 电流调节的偏压电阻，经过 11 KΩ 电阻后接地

3.2.3 PDI 接口

ESC 芯片的应用数据接口称为过程数据接口（Process Data Interface）或物理设备接口（Physical Device Interface），简称 PDI。ESC 提供两种类型的 PDI 接口：

- 直接 IO 信号接口 无需应用层微处理器，最多 32 位引脚；
- DPRAM 数据接口 使用外部微处理器访问，支持并行和串行两种方式。

ET1100 的 PDI 接口类型和相关特性由寄存器 0x0140: 0x0141 进行配置，如表 3.9 所列。

表 3.9 PDI 接口配置

地 址	位	名 称	描 述	复位值
0x0140:	0~7	PDI 类型， 过程数据接口或 物理数据接口	0: 接口无效 4: 数字量 I/O 5: SPI 从机 8: 16 位异步微处理器接口 9: 8 位异步微处理器接口 10: 16 位同步微处理器接口 11: 8 位同步微处理器接口	上电后装载 EEPROM 地 址 0 的数据
0x0141	8	设备状态模拟	0: AL 状态必须由 PDI 设置 1: AL 状态寄存器自动设为 AL 控制寄存器的值	
	9	增强的链接检测	0: 无 1: 使能	
	10	分布时钟同步输 出单元	0: 不使用 (节能) 1: 使能	
	11	分布时钟锁存输 入单元	0: 不使用 (节能) 1: 使能	
	12~15	保留		

PDI 配置寄存器 0x0150 以及扩展 PDI 配置寄存器 0x0152~0x0153 的设置取决于所选择的 PDI 类型, Sync/Latch 接口的配置寄存器 0x0151 与所选用的 PDI 接口无关。

1. 数字量 I/O 接口

数字量 I/O 接口由 PDI 控制寄存器 $0x140 = 4$ 设置, 其信号如图 3.10 所示和表 3.10 所列。它支持不同的信号形式, 通过寄存器 0x150~0x153 可以实现多种不同的配置。

接口信号中除 I/O[31:0]以外的信号称为控制/状态信号, 它们分配在引脚 PDI[39:32]。如果从站使用了两个以上的物理通信端口, PDI[39:32]不能用作 PDI 信号, 即控制/状态信号无效。此时, 可以通过配置引脚 CTRL_STATUS_MOVE 分配 PDI[23:16]或 PDI[15:8]作为控制/状

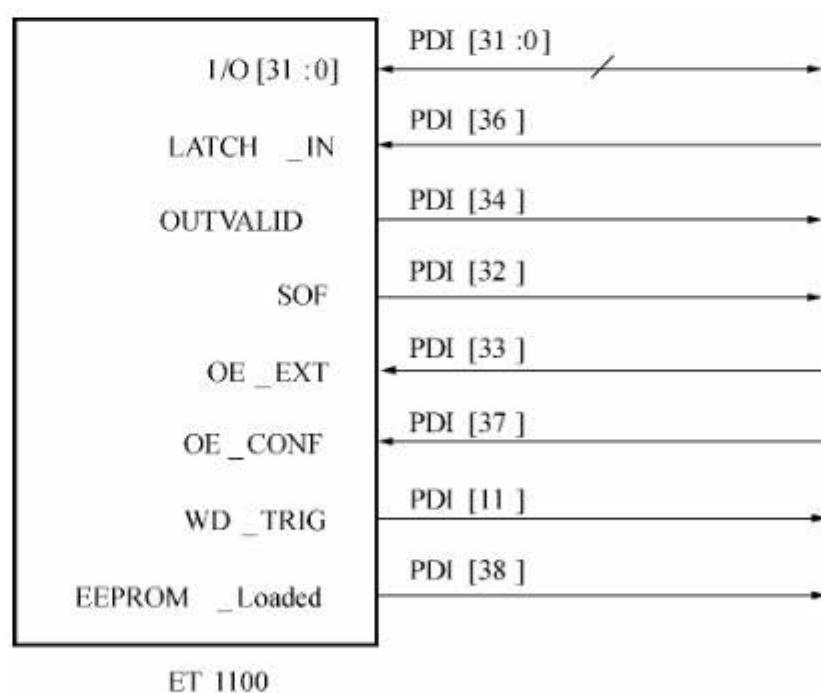


图 3.7 数字量输入/输出信号

态信号。

表 3.2 数字量输入/输出信号描述

信 号	方 向	描 述	信号极性
I/O[31:0]	IN/OUT/BIDIR	输入/输出或双向数据	
LATCH_IN	IN	外部数据锁存信号	高有效
OUTVALID	OUT	输出数据有效/输出事件	高有效
SOF	OUT	帧起始	高有效
OE_EXT	IN	输出使能	高有效
OE_CONF	IN	输出使能配置	
WD_TRIG	OUT	看门狗触发	高有效
EEPROM_Loaded	OUT	PDI 有效, EEPROM 数据正确装载	高有效

数字量输入和输出数据在 ET1100 存储空间的映射地址如表 3.11 所列。主站和从站通过 ECAT 帧和 PDI 接口分别读写这些存储地址来操作数字输入输出信号。

表 3.3 数字量输入/输出信号对应存储地址

地 址	位	描 述	复位值
0x0F00: 0x0F03	0~31	数字量 I/O 输出数据	0
0x1000: 0x1003	0~31	数字量 I/O 输入数据	0

数字量输入/输出接口有以下几种应用和控制方式。

(1) 数字量输入

数字量输入保存在 ESC 过程存储区的 0x1000~0x1003。使用低字节在前模式, I/O[31:0] 对应地址 0x1000~0x1003。ESC 可以使用四种方式采样数字量输入值:

- 由每个帧起始位触发输入采样, 主站读 0x1000~0x1003 命令可以读到同一数据帧的起始位触发的输入采样值;
- 采样时间由外部输入信号 LATCH_IN 控制, ESC 在每个 LATCH_IN 信号的上升沿采样输入数据;
- 由分布时钟的 SYNC0 事件触发输入采样;
- 由分布时钟的 SYNC1 事件触发输入采样。

使用分布时钟同步信号时, 必须使能 SYNC 输出 (寄存器 0x0981), 但是并不一定要输出 SYNC 信号。由于数字 I/O 接口模式下不能应答 SYNC 信号, 因此 SYNC 脉冲宽度不允许设为 0。

(2) 数字量输出

数字量输出数据由 ESC 寄存器 0x0F00~0x0F03 控制, 使用低字节在前模式, 寄存器 0x0F00: 0x0F03 对应 I/O[31:0]。数字量输出值直接产生芯片输出, 具有很快的时间响应。数字量输出刷新可以配置为以下三种形式之一:

- ECAT 数据帧对寄存器 0x0F00~0x0F03 中一个字节的写操作触发;
- 由分布时钟的 SYNC0 事件触发;
- 由分布时钟的 SYNC1 事件触发。

使用分布时钟同步信号时，必须使能 SYNC 输出（寄存器 0x0981），但是并不一定要输出 SYNC 信号。由于数字 I/O 接口模式下不能应答 SYNC 信号，因此 SYNC 脉冲宽度不允许设为 0。

OUTVALID 引脚提供输出刷新指示脉冲信号，即使输出数据保持不变，此信号也出现。当输出使能信号 OE_EXT 为高时，才能在 IO 引脚获得输出信号。

(3) 双向模式

在双向模式下，所有的数据信号是双向的，输入信号通过串联电阻（推荐阻值为 $4.7\text{ k}\Omega$ ）接入 ESC。输出信号由 ESC 驱动，可以使用 OUTVALID 锁存输出数据。双向模式的原理如图 3.11 所示。

本模式仍然可以采用前面介绍的几种方式控制输入采样和输出刷新，注意避免输入输出的双向同时触发，否则会产生错误的输出数据。

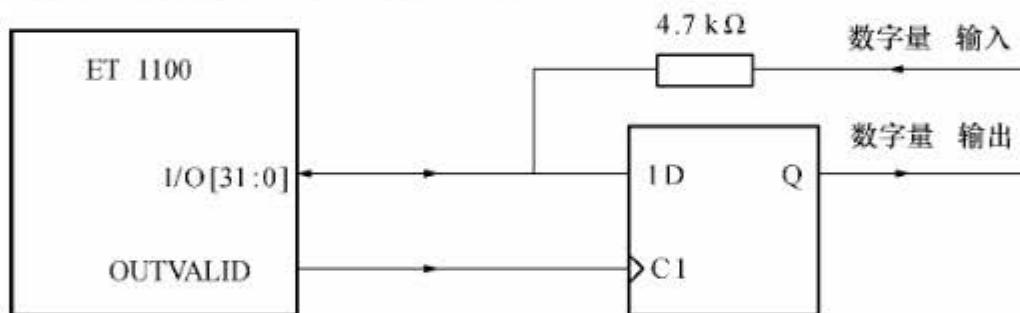


图 3.8 数字 I/O 双向模式原理

使用 OE_EXT 和 OE_CONF 引脚可以控制 IO 引脚状态。OE_EXT 引脚设为低或同步管理器看门狗过期后的 IO 引脚状态如下：

- OE_CONF 为低电平，I/O 引脚电平被拉低；
- OE_CONF 为高电平，I/O 引脚处于高阻态。

(4) EEPROM_Loaded

在 EEPROM 正确装载之后，EEPROM_Loaded 信号指示数字量 I/O 接口可操作。使用时需要外接一个下拉电阻。

2.SPI 从站接口

PDI 控制寄存器 $0x140 = 0x05$ 时，ET1100 使用 SPI 接口。它作为 SPI 从机由带有 SPI 接口的微处理器操作。SPI 接口信号如图 3.12 所示。表 3.12 详细描述了 SPI 接口的各个信号。

由于 SPI 接口占用的 PDI 引脚较少，剩余的 PDI 引脚可以作为通用 IO 引脚使用，包括 16 个通用数字量输入引脚 GPI（General Purpose Input）和 16 个通用数字量输出引脚 GPO（General Purpose Output），如表 3.5 所列。通用数字量输入引脚对应寄存器 0x0F18~0xF1F，通用数字量输出引脚对应寄存器 0x0F10~0xF17。PDI 接口和 ECAT 帧都可以访问这些寄存器，这些引脚以非同步的刷新方式工作。

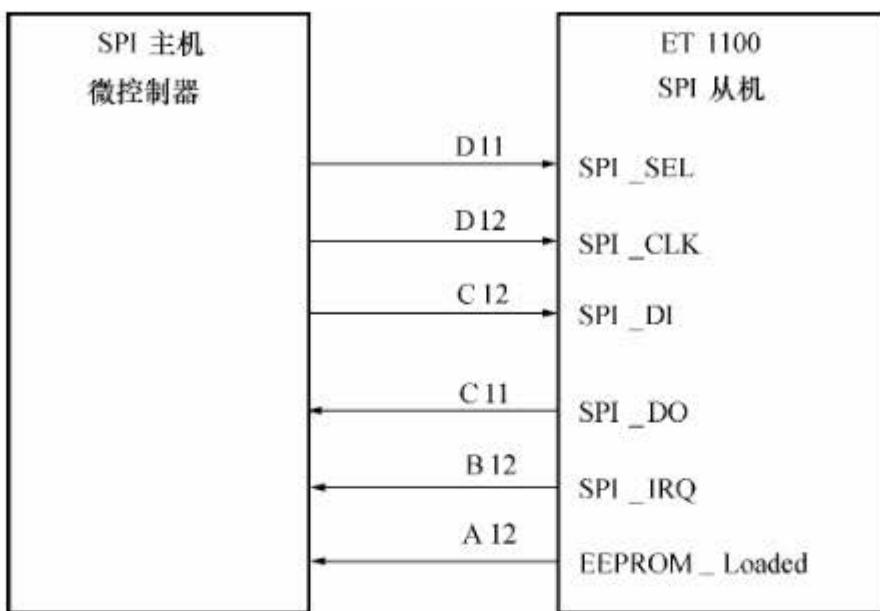


图 3.9 SPI 主机和从机互连图

表 0.4 SPI 接口信号

信 号	方 向	描 述	信号极性
SPI_SEL	IN	SPI 片选	典型配置：低有效
SPI_CLK	IN	SPI 时钟	
SPI_DI	IN	SPI 数据 MOSI	高有效
SPI_DO	OUT	SPI 数据 MISO	高有效
SPI_IRQ	OUT	SPI 中断	典型配置：低有效
EEPROM_Loaded	OUT	PDI 活动，EEPROM 正确装载	高有效

3. 并行微处理器总线接口

并行微处理器总线接口使用复用的地址和数据总线，双向数据总线可选 8 位或 16 位，可以使用同步或异步两种总线操作模式，其接口信号如表 3.13 所列。详细的 PDI 类型选择由寄存器 0x140 设定，如表 3.9 所列。图 3.13、3.14 和 3.15 分别给出了几种 PDI 与微处理器连接示例。

表 3.13 同步/异步微处理器接口信号

同步信号	异步信号	方 向	描 述	信号极性
CPU_CLK_IN	N/A	IN	接口时钟	
CS	CS	IN	片选	典型配置：低有效
ADR[15:0]	ADR[15:0]	IN	地址总线	典型配置：高有效
BHE	BHE	IN	高字节有效（只用于 16 位微处理器）	典型配置：低有效
TS	RD	IN	TS：传输周期启动 RD：读操作	典型配置：低有效

续表 3.13

同步信号	异步信号	方 向	描 述	信号极性
RD/nWR	WR	IN	读或写操作	
DATA[15:0]	DATA[15:0]	双向	16 位微处理器数据总线	高有效
DATA[8:0]	DATA[8:0]	双向	8 位微处理器数据总线	高有效
TA	BUSY	OUT	传输应答	典型配置: 低有效
IRQ	IRQ	OUT	中断	典型配置: 低有效
EEPROM_Loaded	EEPROM_Loaded	OUT	PDI 活动, EEPROM 正确装载	高有效

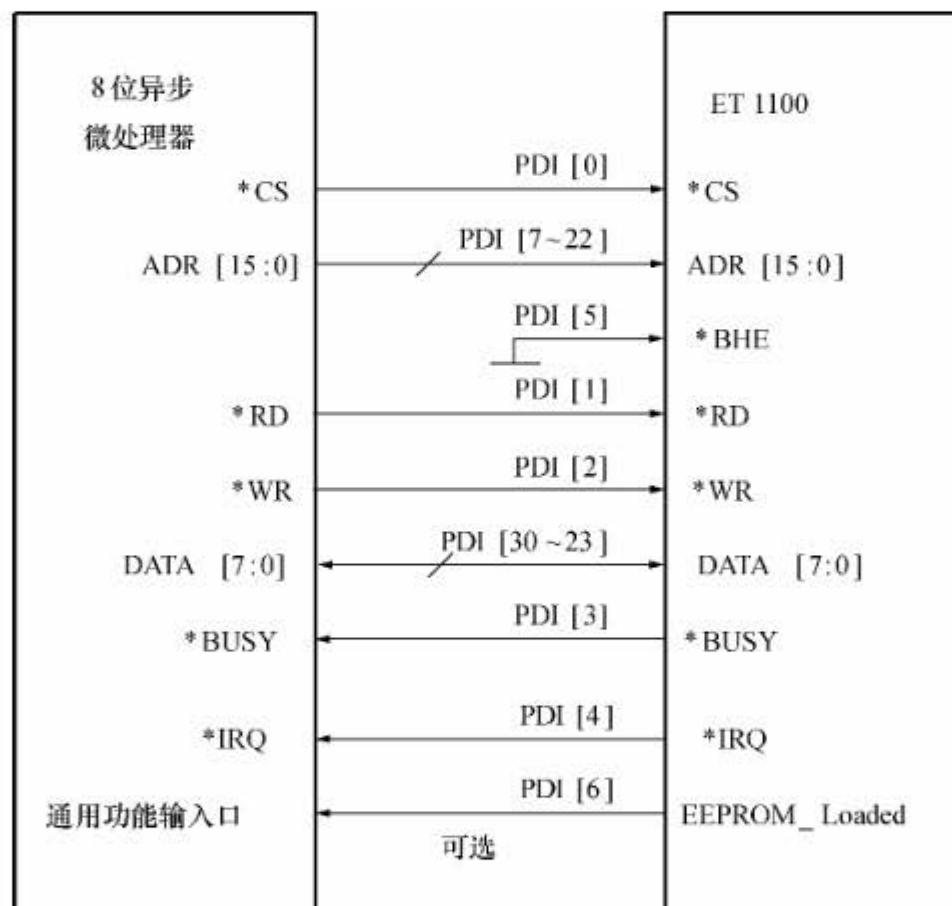


图 3.10 8 位异步微处理器与 ET1100 的数据接口

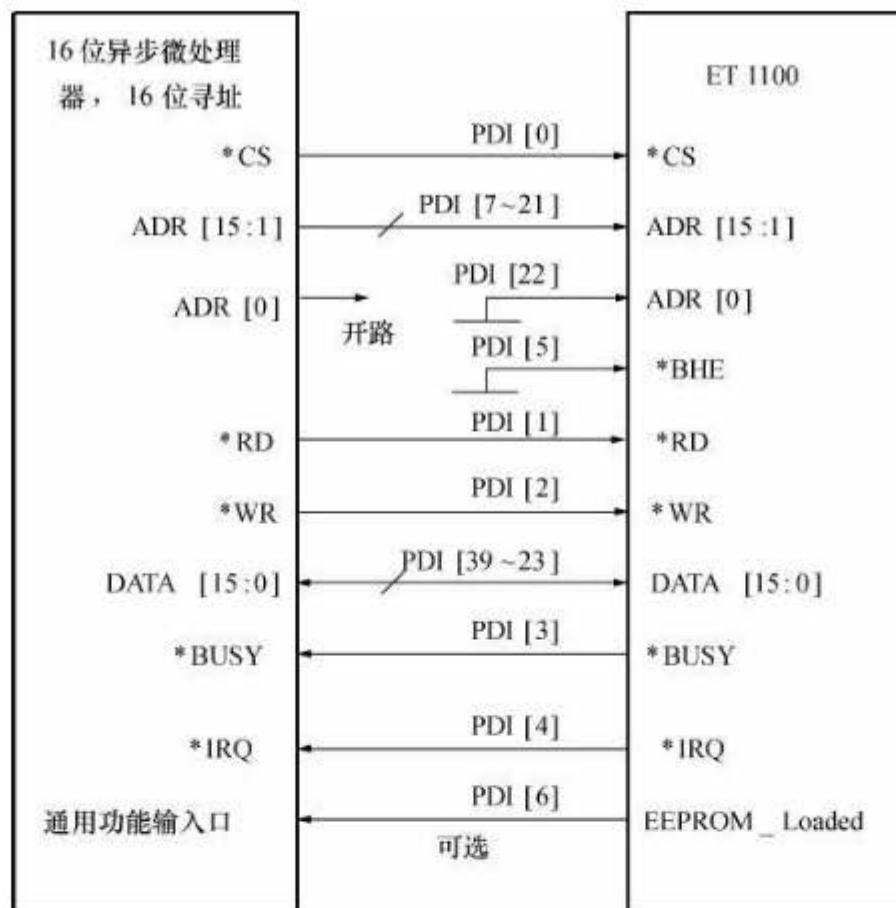


图 3.11 16位异步微处理器与 ET1100 的数据接口

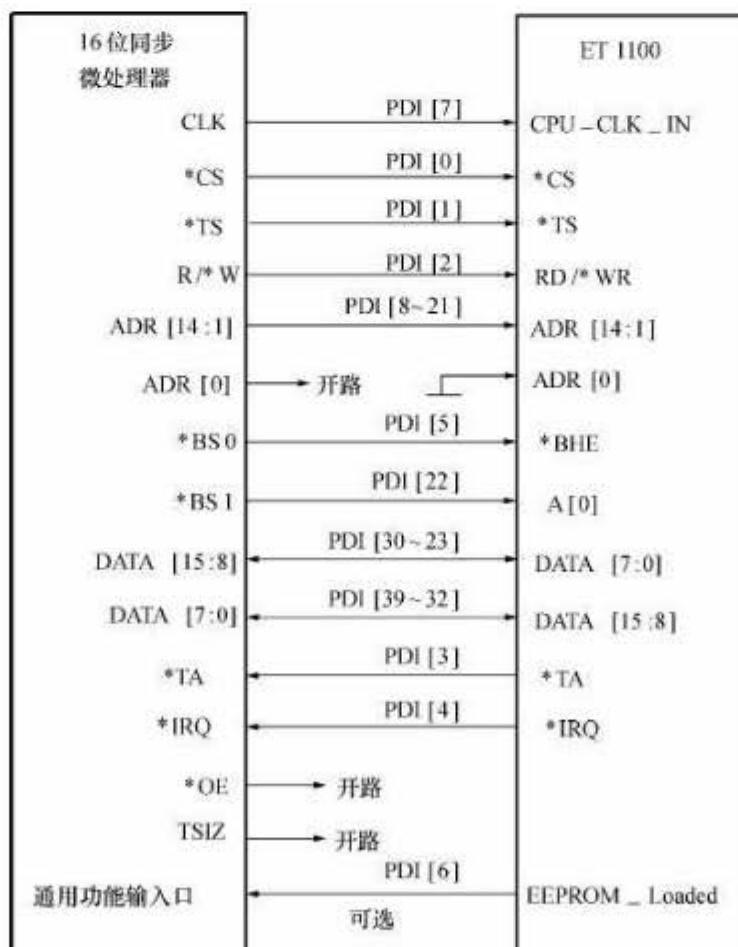


图 3.12 16位同步微处理器使用字节选择信号操作 ET1100

4. 物理通信端口引脚和 PDI 引脚的配置

由 ET1100 引脚定义表 3.3 可知，对 PDI 功能引脚与物理端口 2 和 3 的引脚进行复用，可以获得芯片规模和功能的最佳组合。端口 0 和 1 的引脚与 PDI 引脚无关。表 3.14 总结了 PDI 类型和通信端口使用的组合配置情况。

表 3.5 PDI 和通信端口组合配置

PDI 配置 端口使用配置	异步微 处理器	同步微 处理器	SPI	数字量 IO CTLR_STATUS_MOVE	
				0	1
2 个端口（0 和 1）或 3 个端口，并且端口 2 使用 EBUS	8 位或 16 位	8 位或 16 位	SPI+ 32 位 GPIO	32 位 I/O+控制/状态信号	
3 个端口，都使用 MII 接口	8 位	8 位	SPI+ 24 位 GPIO	32 位 I/O	24 位 I/O+控制/状态信号
4 个端口，至少两个 EBUS 接口	不可用	不可用	SPI+ 16 位 GPIO	24 位 I/O+控制/状态信号	
4 个端口，其中 3 个 MII 接口，一个 EBUS	不可用	不可用	SPI+ 16 位 GPIO	24 位 I/O	16 位 I/O+控制/状态信号
4 个 MII 端口	不可用	不可用	SPI+ 8 位 GPIO	16 位 I/O	8 位 I/O+控制/状态信号

3.2.4 配置引脚

配置引脚在上电时作为输入由 ET1100 锁存配置信息。上电之后这些引脚都有分配的操作功能，必要时引脚信号方向也可以改变。RESET 引脚信号指示上电配置的完成。所有的配置引脚如表 3.15 所列。

这些引脚外接上拉或下拉电阻。外接下拉电阻时，配置信号为 0；使用上拉电阻时，配置信号为 1。EEPROM_SIZE/RUN、P_CONF[0~3]/LinkAct(0~3)等配置引脚也可以用做状态输出引脚来外接 LED，LED 的极性取决于需要配置的值。如果配置数据为 1，需要上拉，引脚输出为 0（低）时发光二极管导通，如图 3.16(a)所示。如果配置数据为 0，引脚需要下拉，引脚输出为 1（高）时发光二极管导通，如图 3.16(b)。

表 3.6 ET1100 配置引脚

描述	配置信号	引脚编号	寄存器映射	设定值
端口模式	P_MODE[0]	L2	0x0E00[0]	00=两个端口(0和1) 01=3个端口(0、1和2) 10=3个端口(0、1和3) 11=4个端口(0、1、2和3)
	P_MODE[1]	M1	0x0E00[1]	
端口配置	P_CONF[0]	J12	0x0E00[2]	0=EBUS
	P_CONF[1]	L1	0x0E00[3]	1=MII
	P_CONF[2]	E3	0x0E00[4]	
	P_CONF[3]	C2	0x0E00[5]	
CPU时钟输出模式, PDI[7]/ CPU_CLK	CLK_MODE[0]	J11	0x0E00[6]	00=off 01=25 MHz 10=20 MHz 11=10 MHz
	CLK_MODE[0]	K2	0x0E00[7]	
TX相位偏移	C25_SHI[0]	L7	0x0E01[0]	00=无MII TX信号延迟 01=MII TX信号延迟10 ns 10=MII TX信号延迟20 ns 11=MII TX信号延迟30 ns
	C25_SHI[1]	M7	0x0E01[1]	
CLK25OUT2输出使能	C25_ENA	L8	0x0E01[2]	0=不使能 1=使能
透明模式使能	TRANS_MODE_ENA	L3	0x0E01[3]	0=常规模式 1=使能透明模式
I/O控制/状态信号转移	CTRL_STATUS_MOVE	E2	0x0E01[4]	0=I/O无控制状态引脚转移 1=I/O控制状态引脚转移
PHY地址偏移	PHYAD_OFF	C3	0x0E01[5]	0=PHY地址使用1~4 1=PHY地址使用17~20
链接有效信号极性	LINKPOL	K11	0x0E01[6]	0=LINK_MII(x)低有效 1=LINK_MII(x)高有效
保留	RESERVED	B2	0x0E01[7]	
EEPROM容量	EEPROM SIZE	H11	0x0502[7]	0=单字节地址(16Kbit) 1=双字节地址(32Kbit到4 Mbit)

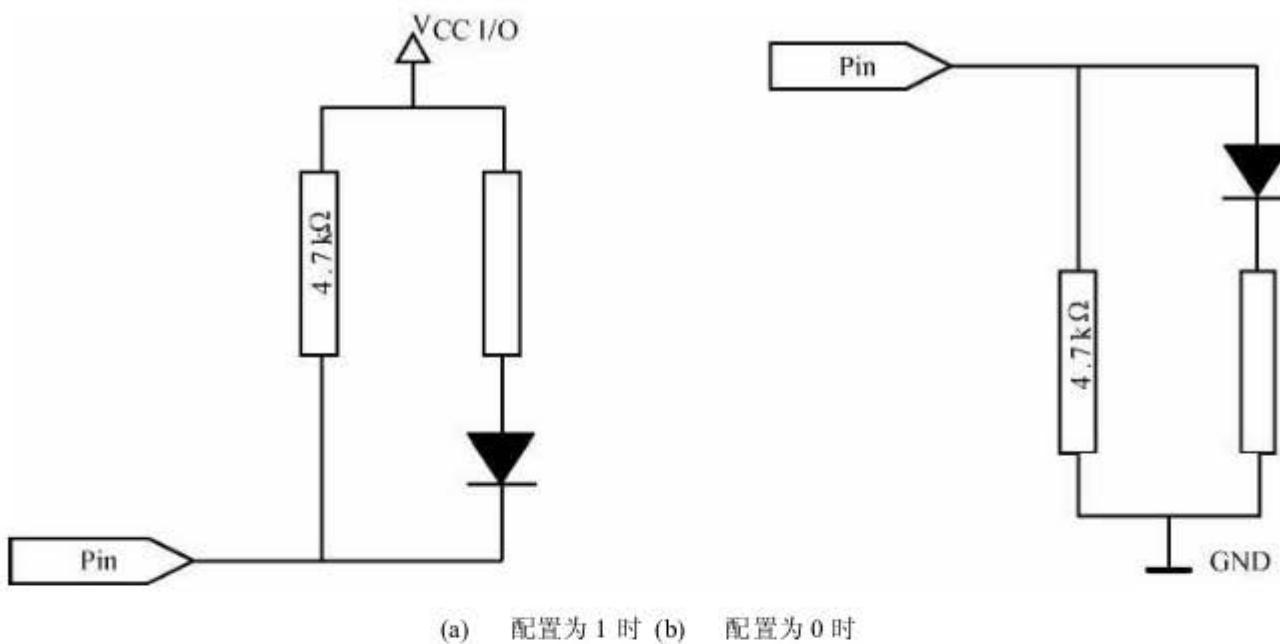


图 3.13 LED 双功能引脚连接

3.2.5 其它引脚

表 3.16 列出了 ET1100 的其他功能引脚，其时钟源引脚的应用见 3.2.2 小节的“1. MII 接口”部分，本节对复位和供电引脚做详细介绍。

表 3.7 ET1100 其他引脚

功 能	引脚名称	编 号	方 向	功 能
时钟源	OSC_IN	G12	I	时钟源输入，外接晶体或振荡器
	OSC_OUT	F12	O	时钟源，外接晶体
EEPROM 接口	EEPROM_CLK	G11	BD	EEPROM 接口 I ² C 通信时钟
	EEOROM_DATA	F11	BD	EEPROM 接口 I ² C 通信数据
MII 管理	MII_CLK/LINKPOL	K11	BD	PHY 管理接口时钟
	MII_DATA	K12	BD	PHY 管理接口数据
复位信号	RESET	H12	BD	内部复位状态输出信号或外部复位控制输入
测试模式	TESTMODE	H3	I	为测试保留，应该接地
供电引脚	见表 3.6			供 电

1. 复位引脚

集电极开路输入/输出 RESET 表示 ET1100 的复位状态，以下三种情况可以引起 ET1100 内部复位：

- ① 在上电之后进入复位状态；
- ② 供电电压过低；
- ③ 由写复位寄存器 0x0040 发起一次复位。

在三个连续的数据帧中向寄存器 0x0040 写入 0x52('R')、0x45('E')和 0x53('S')之后，

触发一次复位。读出值表示了复位的过程，写入 0x52 后读到 0x01，写入 0x45 和 0x53 后读到 0x02，其余为 0。

内部复位时 RESET 输出信号可以用于复位其他外围芯片，例如以太网 PHY 芯片。RESET 引脚由外部设备拉低时 ET1100 也进入复位状态。RESET 引脚连接如图 3.17 所示。

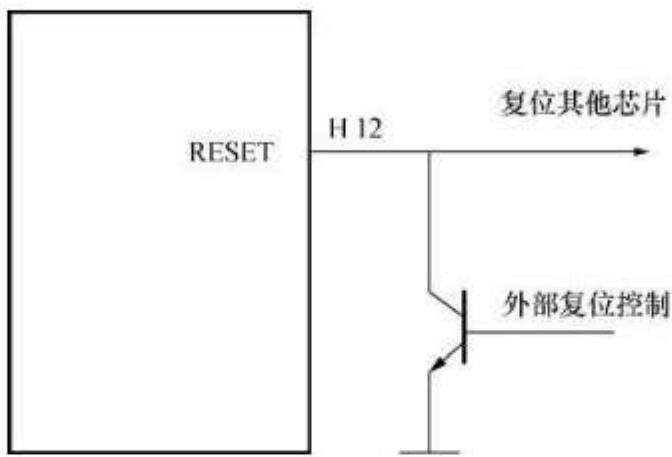


图 3.14 RESET 引脚连接图

2. ET1100 供电引脚

表 3.6 列出了 ET1100 的全部供电引脚，共包括三种类型（如图 3.18 所示）：

(1) I/O 信号电源 $V_{CC_{I/O}}$

$V_{CC_{I/O}}$ 的电压直接决定所有输入和输出信号的电平，它可以使用 3.3 V 或 5 V 供电。使用 3.3 V 供电时，I/O 信号电平为 3.3 V，不允许使用 5 V 输入。使用 5 V 供电时，I/O 信号电平即为 5 V。所有的 $V_{CC_{I/O}}$ 和 $GND_{I/O}$ 引脚都要连接，且必须并联滤波电容稳定电压。

(2) 逻辑内核电源 $V_{CC_{Core}}$

ET1100 的逻辑内核要求 2.5 V 供电，既可以由内部 LDO(Low Dropout Regulator，低压差线性稳压器) 产生，也可以外供。内部 LDO 使用 $V_{CC_{I/O}}$ 作为电源。在这两种情况下 $V_{CC_{Core}}/GND_{Core}$ 之间必须并联滤波电容稳定电压。

当外供内核电压 ($V_{CC_{Core}}$) 高于内部 LDO 输出电压时 LDO 停止供电。所以使用外部

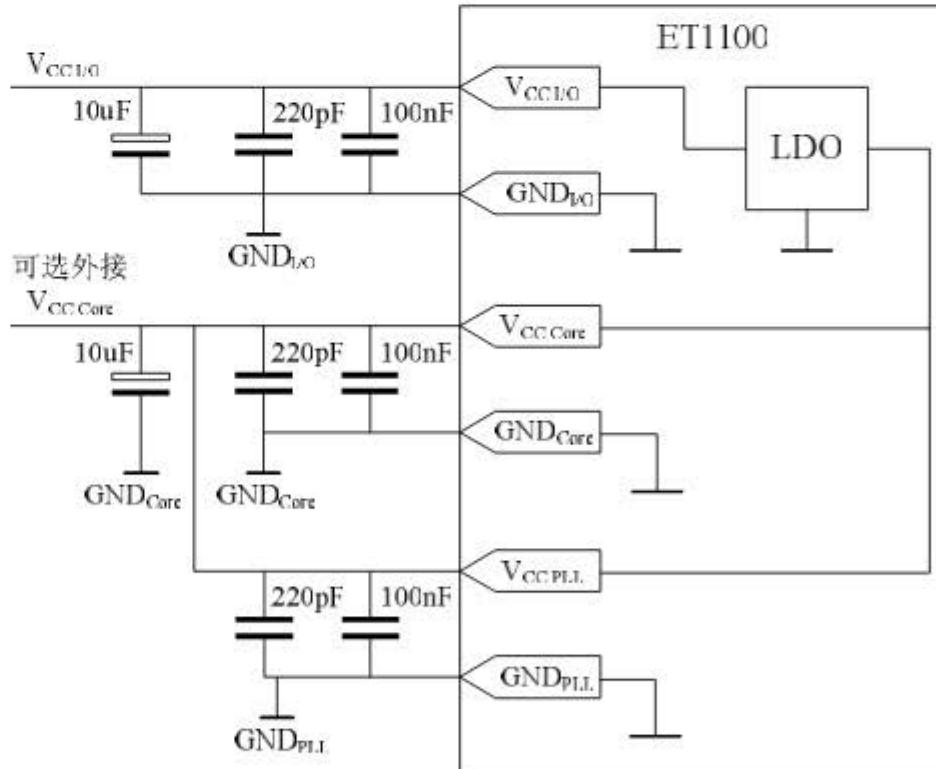


图 3.15 ET1100 供电连接

电源提供 $V_{CC\ Core}$ 时，必须比内部 LDO 额定输出电压至少高出 0.1 V，以使 LDO 停止输出。使用内部 LDO 会增加 ET1100 耗电。

(3) PLL (Phase Locked Loop) 电源 $V_{CC\ PLL}$

$V_{CC\ PLL}$ 与 $V_{CC\ Core}$ 功能相同，其连接如图 3.18 所示。

3.3 ESC 数据链路控制

3.3.1 ESC 数据帧处理

每个 ESC 可以最多支持 4 个数据收发端口，每个端口都可以处在打开或闭合状态。如果端口打开，则可以向其它 ESC 发送数据帧或从其它 ESC 接收数据帧。一个闭合的端口不会与其它 ESC 交换数据帧，它在内部将数据帧转发到下一个逻辑端口，直到数据帧到达一个打开的端口。如图 3.19 所示。

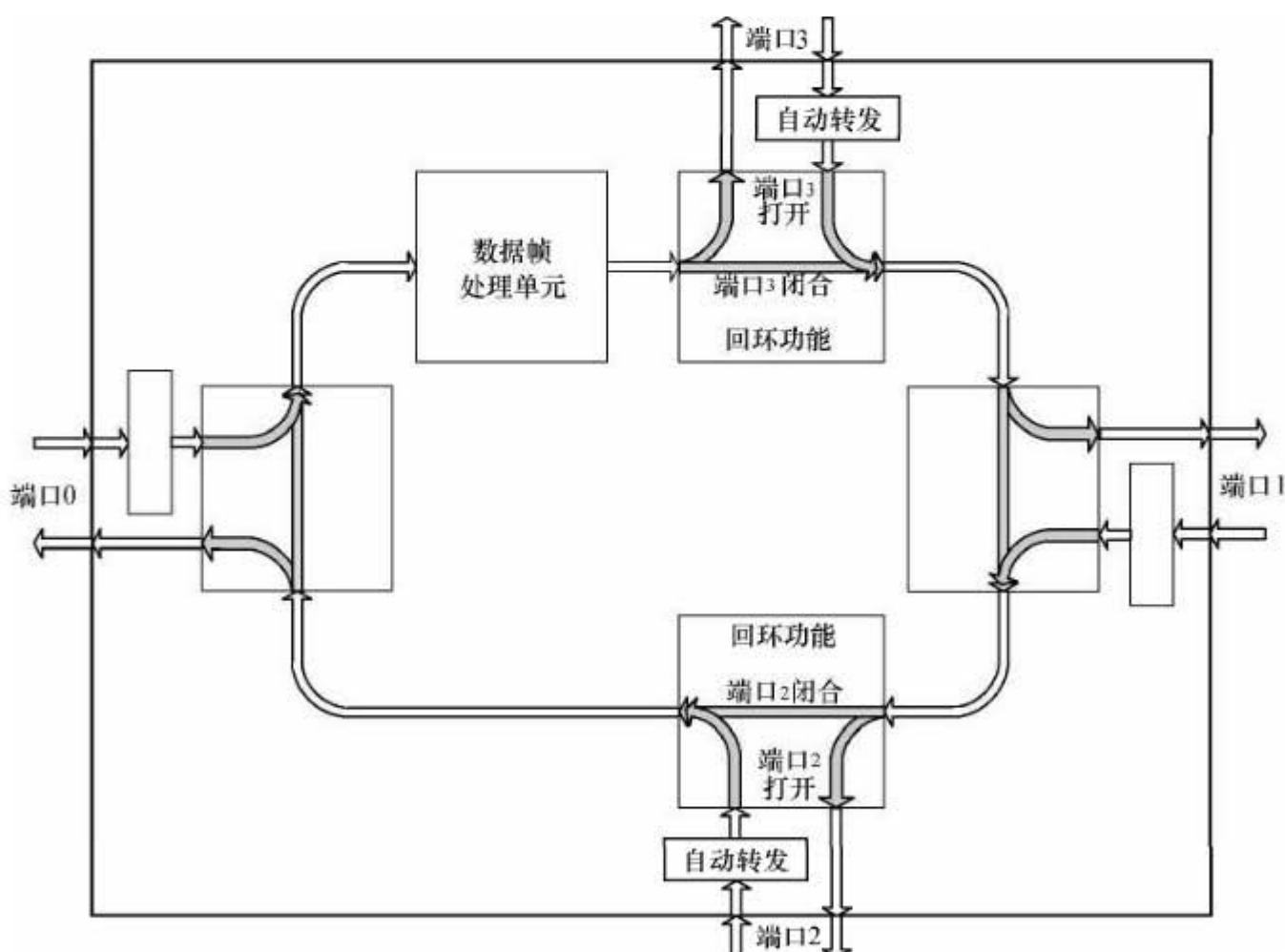


图 3.16 ESC 内部数据帧传输顺序

数据帧在 ESC 内部的处理顺序取决于所使用的端口数目，如表 3.17 所列。在 ESC 内部经过数据帧处理单元的方向称为“处理”方向，其他方向称为“转发”方向。

表 3.8 数据帧处理顺序

端口数	帧处理顺序
2	0→数据帧处理单元→1/1→0
3	0→数据帧处理单元→1/1→2/2→0 或 0→数据帧处理单元→3/3→1/1→0
4	0→数据帧处理单元→3/3→1/1→2/2→0

ESC 支持 EtherCAT、UDP/IP 和 VLAN (Virtual Local Area Network) 数据帧类型。ESC 能处理包含 EtherCAT 数据子报文的 EtherCAT 数据帧和 UDP/IP 数据帧，也能处理带有 VLAN 标记的数据帧，此时 VLAN 设置被忽略而 VLAN 标记不被修改。

由于 ET1100、ET1200 和 EtherCAT 从站没有 MAC 地址和 IP 地址，它们只能支持直连模式或使用管理型的交换机实现开放模式，由交换机中的端口地址来识别不同的 EtherCAT 网段。

ESC 修改了标准以太网的数据链路 DL (Data Link)，数据帧由 ESC 直接转发处理，从而获得最小的转发延时和最短的周期时间。为了降低延迟时间，ESC 省略了发送 FIFO。但是，为了隔离接收时钟和处理时钟，ESC 使用了接收 FIFO (RX FIFO)。RX FIFO 的大小取决于数据接收方和数据发送方的时钟源精度，以及最大的数据帧字节数。主站可以通过设置数据链路 DL 控制寄存器 (0x0100~0x0103) 的位 16~18 来调整 RX FIFO (如表 3.18 所列)，但是不允许完全取消 RX FIFO。默认的 RX FIFO 可以满足最大的以太网数据帧和 100 ppm 的时钟源精度。使用 25 ppm 的时钟源精度时可以将 RX FIFO 设置为最小。

ESC 的转发延时由 RX FIFO 大小和 ESC 数据帧处理单元延迟决定，而 EtherCAT 从站的数据帧传输延时还与它所使用的物理层器件相关，使用 MII 接口时，由于 PHY 芯片的接收和发送延时较大，一个端口的传输延时约为 500 ns；使用 EBUS 接口时，延时较小，通常约为 100 ns，但是，EBUS 最大只能传输 10 m 的距离。

3.3.2 ESC 通信端口控制

端口的回路状态可以由主站写数据链路 DL 控制寄存器 (0x0100~0x0103) 来控制。ESC 支持强制回路控制（不管连接状态如何都强制打开或闭合），以及自动回路控制（由每个端口的连接状态决定打开或闭合）。在自动模式下，如果建立连接则端口打开，如果失去连接则端口闭合。端口失去连接而自动闭合，再次建立连接后，它必须被主动打开，或者端口收到有效的以太网数据帧后也可以自动打开。ESC 端口的状态可以从 DL 状态寄存器 (0x0110~0x0111) 中读取。DL 控制寄存器和 DL 状态寄存器的详细定义如表 3.18 所列。

表 3.9 ESC 数据链路层控制和状态

地 址	位	名 称	描 述	复位值
0x0100~0x0103	0~31	DL 控制	ESC 数据链路层通信控制	
	0	转发规则	0：处理 EtherCAT 数据帧，非 EtherCAT 帧只转发不处理 1：处理 EtherCAT 数据帧，并设置源 MAC 地址为本地管理地址 (MAC 地址字节 0=0x02) 终结非 EtherCAT 帧	1

续表 3.18

地 址	位	名 称	描 述	复位值
0x0100~0x0103	1	暂时使用 0x101 配置	0: 永久使用 1: 使用大约 1 s, 然后恢复到之前的配置	0
	2~7	保留		
	8: 9	端口 0 环路控制	00: 自动 链路断开时闭合, 链路连接时打开	00
	10:11	端口 1 环路控制		00
	12:13	端口 2 环路控制	01: 自动闭合 链路断开时闭合, 链路连接时写入 0x01 后打开	00
	14:15	端口 3 环路控制	10: 无论链路状态如何, 都打开 11: 无论链路状态如何, 都闭合	00
	16:18	RX FIFO 大小	ESC 在 FIFO 至少半满之后才开始转发数据帧, RX 延时减小 Value EBUS MII 0: -50ns -40ns 1: -40ns -40ns 2: -30ns -40ns 3: -20ns -40ns 4: -10ns 无变化 5: 无变化 无变化 6: 无变化 无变化 7: 缺省值 缺省值	7
	19	EBUS 抖动	0: 正常抖动 1: 降低抖动	0
	20~23	保留		0
	24	站点别名	0: 忽略站点别名 1: 所有配置地址寻址使用别名	0
	25~31	保留		0
0x0110~0x0111	0~15	DL 状态	ESC 数据链路状态	
	0	PDI 可操作	0: EEPROM 没有装载, PDI 不可操作, 不可访问过程数据存储区 1: EEPROM 正确装载, PDI 可操作, 可以访问过程数据存储区	0
	1	PDI 看门狗状态	0: 看门狗过期 1: 看门狗已重装载	0
	2	增强的链接检测功能	0: 没有激活 1: 激活	EEPROM
	3	保留		0

续表 3.18

地 址	位	名 称	描 述	复位值
0x0110~0x0111	4	端口 0 物理链接	0: 无链接 1: 检测到链接 MII: 对应于 LINK_MII 信号 EBUS: 链接检测的结果	0
	5	端口 1 物理链接		0
	6	端口 2 物理链接		0
	7	端口 3 物理链接		0
	8	端口 0 环路状态	0: 打开, 数据帧在此端口离开本 ESC 1: 闭合, 数据帧被转发到内部的下一端口	0
	9	端口 0 通信	0: 无稳定的通信 1: 建立了通信	0
	10	端口 1 环路状态	0: 打开, 数据帧在此端口离开本 ESC 1: 闭合, 数据帧被转发到内部的下一端口	0
	11	端口 1 通信	0: 无稳定的通信 1: 建立了通信	0
	12	端口 2 环路状态	0: 打开, 数据帧在此端口离开本 ESC 1: 闭合, 数据帧被转发到内部的下一端口	0
	13	端口 2 通信	0: 无稳定的通信 1: 建立了通信	0
	14	端口 3 环路状态	0: 打开, 数据帧在此端口离开本 ESC 1: 闭合, 数据帧被转发到内部的下一端口	0
	15	端口 3 通信	0: 无稳定的通信 1: 建立了通信	0

通信端口由主站控制, 从站微处理器不操作数据链路。端口被使能, 而且满足以下任一条件时, 端口将被打开:

- DL 控制寄存器中端口设置为自动时, 端口上有活动的连接;
- DL 控制寄存器中回路设置为自动闭合时, 端口上建立连接, 并且向寄存器 0x0100 相应控制位再次写入 0x01;
- DL 控制寄存器中回路设置为自动闭合时, 端口上建立连接, 并且收到有效的以太网数据帧;
- DL 控制寄存器中回路设置为常开。

满足以下任一条件时, 端口将被闭合:

- 端口不可用, 或没有被使能;
- DL 控制寄存器中回路设置为自动时, 端口上没有活动的连接;
- DL 控制寄存器中回路设置为自动闭合时, 端口没有活动的连接, 或者建立连接后没有向相应控制位再次写入 0x01;
- DL 控制寄存器中回路设置为常闭。

在所有的端口不管是强制或自动而处于闭合状态时, 端口 0 都将打开作为恢复端

口。可以通过这个端口实现读/写操作，以便改正 DL 控制寄存器的设置。此时，DL 状态寄存器仍然反映正确的状态。

3.3.3 数据链路错误检测

ESC 在两个功能块中检测 EtherCAT 数据帧错误：自动转发模块和 EtherCAT 数据帧处理单元。自动转发模块能检测到的错误有：

- 物理层错误 (RX 错误);
- 数据帧过长;
- CRC 校验错误;
- 数据帧无以太网起始符 SOF (Start Of Frame)。

EtherCAT 数据帧处理单元可以检测到的错误有：

- 物理层错误 (RX 错误);
- 数据帧长度错误;
- 数据帧过长;
- 数据帧过短;
- CRC 检验错误;
- 非 EtherCAT 数据帧 (如果 0x100.0=1)。

ESC 寄存器中有一些错误指示寄存器用来帮助监测和定位错误，如表 3.19 所列。所有计数器的最大值都为 0xFF，计数达到 0xFF 后停止，不再循环计数，必须由写操作来清除。

表 3.10 数据链路错误计数器

地 址	位	名 称	描 述	复位值
0x0300+i*2	0~7	端口 i 无效帧计数	计数达到 0xFF 后停止，写接收错误计数器 0x0300~0x030B 中的任意一个后清除所有值	0
0x0301+i*2	8~15	端口 i 接收错误计数	计数达到 0xFF 后停止，接收错误直接与 MII 或 EBUS 接口的 RX_ERR 信号相关，写接收错误计数器 0x0300~0x030B 中的任意一个后清除所有值	0
0x0308+i	0~7	端口 i 转发错误计数	计数达到 0xFF 后停止，写接收错误计数器 0x0300~0x030B 中的任意一个后清除所有值	0
0x030C	0~7	数据帧处理单元错误计数	计数经过处理单元的错误帧，比如 FCS 错误、子报文结构错误等，计数达到 0xFF 后停止，写操作清除其中的值	0
0x030D	0~7	PDI 错误计数计数	计数 PDI 操作出现的接口错误，达到 0xFF 后停止，写操作清除其中的值	
0x0310+i	0~7	端口 i 链接丢失计数	计数达到 0xFF 后停止，写其中一个寄存器将清除所有值	

3.3.4 ESC 数据链路地址

EtherCAT 协议规定使用设置寻址时，有两种从站地址模式，表 3.20 列出了两种设置站点地址时使用的寄存器。

表 3.20 ESC 数据链路地址寄存器

地 址	位	名 称	描 述	复位值
0x0010~ 0x0011	0~15	设 置 站 点 地 址	设置寻址所使用的地址 (FPRD、FPWR 和 FPRW 命令)	0
0x0012~ 0x0013	0~15	设 置 站 点 别 名	设置寻址所使用的地址别名，是否使用这个别名取决于 DL 控制寄存器 0x0100~0x0103 的位 24	0 直到首次载入 EEPROM 地址 0x0004 中的数据

(1) 由主站在数据链路启动阶段配置给从站

主站在初始化状态时，使用 APWR 命令写从站寄存器 0x0010~0x0011，为从站设置一个与连接位置无关的地址，在后续的运行过程中即可使用此地址访问从站。

(2) 由从站在上电初始化的时候从自身的配置数据存储区装载

每个 ESC 芯片都配有 EEPROM 芯片存储配置数据，其中包括一个站点别名。ESC 在上电初始化时自动装载 EEPROM 中的数据，将站点别名装载到寄存器 0x0012~0x0013。主站在链路启动阶段使用顺序寻址命令 APRD 读取各个从站的设置地址别名，并在后续运行中使用。使用别名之前，主站还需要设置 ESC DL 控制寄存器 0x0100~0x0103 的位 24 等于 1，通知从站将使用站点别名进行设置地址寻址。

使用从站别名可以保证即使网段拓扑改变或者添加或取下设备时，从站设备仍然可以使用相同的设置地址。

3.3.5 逻辑寻址控制

EtherCAT 子报文可以使用逻辑寻址方式访问 ESC 内部存储空间，ESC 使用 FMMU 通道实现逻辑地址的映射。逻辑寻址和 FMMU 的原理参见“2.3.3 逻辑寻址和 FMMU”小节。每个 FMMU 通道使用 16 个字节配置寄存器，从 0x0600 开始。表 3.21 详细解释了 FMMU 通道配置寄存器的含义。

表 3.11 FMMU 通道配置寄存器

偏移地址	位	名 称	描 述	复位值
+0x0: 0x3	0~31	数 据 逻 辑 起 始 地 址	在 EtherCAT 地址空间内的逻辑起始地址	0
+0x4: 0x5	0~15	数 据 长 度 (字 节 数)	从第一个逻辑 FMMU 字节到最后一个 FMMU 字节的偏移+1，例如，如果使用了 2 个字节，取值为 2	0

续表 3.21

+0x6	0~2	数据逻辑起始位	应该映射的逻辑起始位, 从最低有效位 (=0) 到最高有效位 (=7) 计数	0
	3~7	保留		
+0x7	0~2	数据逻辑终止位	应该映射的最后一一位, 从最低有效位 (=0) 到最高有效位 (=7) 计数	0
	3~7	保留		
+0x8: 0x9	0~15	从站物理内存起始地址	物理起始地址, 映射到逻辑起始地址	0
+0xA	0~2	物理内存起始位	物理起始位, 映射到逻辑起始位	0
	3~7	保留		
+0xB	0	读操作控制	0: 无读访问映射 1: 使用读访问映射	0
	1	写操作控制	0: 无写访问映射 1: 使用写访问映射	
	2~7	保留		
+0xC	0	激活	0: 不激活 FMMU 1: 激活 FMMU, 用以检查根据映射配置所映射的逻辑地址块	0
	1~7	保留		
+0xD~+0xF	0~23	保留		0

3.4 ESC 应用层控制

3.4.1 状态机控制和状态

表3.22列出了从站状态机控制和状态寄存器。主站和从站按照以下规则执行状态转化:

- (1) 主站要改变从站状态时, 将目的状态写入**从站 AL 控制位 (0x0120.0~3)**;
- (2) 从站读取到新状态请求之后, 检查自身状态:
 - 如果可以转化, 则将新的状态写入**状态机实际状态位 (0x0130.0~3)**;
 - 如果不能转化, 则不改变实际状态位, 设置错误指示位 (0x0130.4), 并将错误码写入 0x0134~0x0135。
- (3) 主站读取状态机实际状态 (0x0130):
 - 如果正常转化, 则执行下一步操作;
 - 如果出错, 主站读取错误码, 并写 AL 错误应答 (0x0120.4) 来清除 AL 错误指示。

表 3.12 ESC 状态机控制和状态寄存器

地址	位	名称	描述	复位值
0x0120: 0x0121	0~3	AL 控制位	发起从站状态机的状态切换 1: 请求初始化状态 2: 请求预运行状态 3: 请求 Bootstrap 状态 4: 请求安全运行状态 8: 请求运行状态	1
	4	AL 错误应答	0: 无错误应答 1: 应答 AL 状态寄存器中的错误	0
	5~15	保留		0
0x0130: 0x0131	0~3	从站状态机实际状态	1: 初始化状态 3: Bootstrap 状态 2: 预运行状态 4: 安全运行状态 8: 运行状态	1
	4	AL 错误指示	0: 从站处在所请求的状态或标志被清除	0
	5~15	保留		0
0x0134: 0x0135	0~15	AL 状态码	应用层状态, 是否有错误, 及错误代码	0

使用微处理器 PDI 接口时, AL 控制寄存器由握手机制操作。ECAT 帧写 AL 控制寄存器后, PDI 必须执行一次, 否则, ECAT 帧不能继续写操作。只有在复位后 ECAT 帧才能恢复写 AL 控制寄存器。PDI 接口为数字量 I/O 时, 从站没有通过外部微处理器读 AL 控制寄存器, 此时, 主站设置设备模拟位 0x0140.8=1 (见表 3.9), ESC 将自动复制 AL 控制寄存器的值到 AL 状态寄存器。AL 状态码的定义如表 3.23 所列。表中 “+E” 表示设置了 AL 错误指示位 (0x0130.4)。

表 3.13 AL 状态码定义

编码	描述	发生错误的状态或状态改变	结果的状态
0x0000	无错误	任意状态	当前状态
0x0001	未知错误	任意状态	任意状态+E
0x0011	无效状态改变请求	I→S, I→O, P→O, O→B, S→B, P→B	当前状态+E
0x0012	未知状态请求	任意状态	当前状态+E
0x0013	不支持引导状态	I→B	I+E
0x0014	固件程序无效	I→P	I+E
0x0015	无效的邮箱配置	I→B	I+E

续表 3.23

编 码	描 述	发生错误的状态或状态改变	结果的状态
0x0016	无效的邮箱配置	I→P	I+E
0x0017	无效 SM 通道配置	P→S, S→O	当前状态+E
0x0018	无有效的输入数据	O, S, P→S	P+E
0x0019	无有效的输出数据	O, S→O	S+E
0x001A	同步错误	O, S→O	S+E
0x001B	SM 看门狗	O, S	S+E
0x001C	无效的 SM 类型	O, S P→S	S+E P+E
0x001D	无效的输出配置	O, S P→S	S+E P+E
0x001E	无效的输入配置	O, S, P→S	P+E
0x001F	无效的看门狗配置	O, S, P→S	P+E
0x0020	从站需要冷启动	任意状态	当前状态+E
0x0021	从站需要 Init 状态	B, P, S, O	当前状态+E
0x0022	从站需要 Pre-Op	S, O	S+E, O+E
0x0023	从站需要 Safe-Op	O	O+E
0x0030	无效的 DC SYNC 配置	O, S	S+E
0x0031	无效的 DC 锁存配置	O, S	S+E
0x0032	PLL 错误	O, S	S+E
0x0033	无效的 DC IO 错误	O, S	S+E
0x0034	无效的 DC 超时错误	O, S	S+E
0x0042	EOE 邮箱通信错误	B, P, S, O	当前状态+E
0x0043	COE 邮箱通信错误	B, P, S, O	当前状态+E
0x0044	FOE 邮箱通信错误	B, P, S, O	当前状态+E
0x0045	SOE 邮箱通信错误	B, P, S, O	当前状态+E
0x004F	VOE 邮箱通信错误	B, P, S, O	当前状态+E

3.4.2 中断控制

ESC 支持两种类型的中断：给本地微处理器的 AL 事件请求中断和给主站的 ECAT 帧中断。另外，分布时钟的同步信号也可以用作微处理器的中断信号。表 3.24 列出了 ESC 中断控制使用的寄存器。

表 3.14 ESC 中断控制寄存器

地 址	位	名 称	描 述	复位值
0x0200: 0x0201	0~15	ECAT 帧中断屏蔽	ECAT 帧中断请求是否映射到状态位, 位定义同 ECAT 帧中断请求寄存器 0x0210: 0x0211	0
0x0204: 0x0207	0~31	AL 事件中断请求屏蔽	AL 事件请求寄存器是否映射到 PDI 中断信号*IRQ (B12), 位定义同 AL 中断请求寄存器 0x0220: 0x0223 0: 相应的中断请求位不映射 1: 相应的中断请求位映射	0x00FF: 0xFF0F
0x0210: 0x0211	0~15	ECAT 帧中断请求		
	0	锁存事件	0: 无新锁存事件 1: 锁存事件发生 读取锁存时间寄存器中的一个字节可清除此位	0
	1	保留		
	2	DL 状态事件	0: DL 状态无改变 1: DL 状态发生变化 通过读 DL 状态寄存器来清除	0
	3	AL 状态事件	0: AL 状态无改变 1: AL 状态发生变化 读 AL 状态寄存器清除	0
	4	每个 SM 状态 镜像值	0: 无同步管理器通道事件 1: 有同步管理器事件发生	0
	5			
	...			
	11			
	12~15	保留		
0x0220: 0x0223	0~15	AL 事件请求		
	0	AL 控制事件	0: AL 控制寄存器无变化 1: 主站写了 AL 控制寄存器	0
	1	锁存事件	0: Latch 输入无变化 1: Latch 输入至少改变一次	0
	2	SYNC0 状态	0x0151.3=1 时有效, 读 SYNC0 状态寄存器 0x098E.0 清除	0
	3	SYNC1 状态	0x0151.4=1 时有效, 读 SYNC1 状态寄存器 0x098F.0 清除	
	4	SM 激活寄存 器变化	0: 无变化 1: 至少一个 SM 发生变化 读 SM 激活寄存器清除	0
	5~7	保留		
	8	SM 状态镜像	SM 通道 0 到 SM 通道 15 状态位映射	
	9		0: 无 SM 通道事件发生 1: 发生 SM 事件等待处理	
	...			
	23			
	24~31	保留		

(1) PDI 中断

所有发生的 AL 事件请求都映射到寄存器 0x0220~0x0223，由事件屏蔽寄存器 0x0204~0x0207 决定哪些事件将触发给微处理器的中断信号 IRQ。微处理器响应中断后，在中断服务程序中读取 AL 事件请求寄存器，根据所发生的事件做相应的处理。

(2) ECAT 帧中断

ECAT 帧中断用来通知 EtherCAT 主站从站发生的 AL 事件。它使用 EtherCAT 子报文头中的状态位（见图 2.6）传输 ECAT 中断请求寄存器 0x0210~0x0211。ECAT 帧通过中断屏蔽寄存器 0x0200~0x0201 来决定哪些事件会被写入状态位并发送给主站。

(3) SYNC 信号中断

同步信号可以映射到 IRQ 信号以触发中断，此时同步引脚可以用作 Latch 输入引脚，IRQ 信号有约 40 ns 的抖动，同步信号只有约 12 ns 的抖动。所以也可以将 SYNC 信号直接接到微处理器的中断引脚，使微处理器快速响应同步信号中断。

3.4.3 看门狗控制

ESC 支持两种内部看门狗：监测过程数据刷新的过程数据看门狗和监测 PDI 运行的 PDI 看门狗。表 3.25 列出了看门狗控制相关寄存器。

(1) 过程数据看门狗

通过设置 SM 控制寄存器（0x0804+Nx8）的位 6 来使能相应的过程数据看门狗。设置过程数据看门狗定时器的值（0x0420~0x0421）为零将使看门狗无效。过程数据缓存区被刷新后，过程数据看门狗将重新开始计数。过程数据看门狗超时后，将触发以下操作：

- 设置过程数据看门狗状态寄存器 0x0440.0 = 0；
- 数字量 I/O PDI 接口收回数字量输出数据，不再驱动输出信号或拉低输出信号；
- 过程数据看门狗超时计数寄存器（0x0442）增加。

(2) PDI 看门狗

一次正确的 PDI 读写操作可以启动 PDI 看门狗重新计数。设置 PDI 看门狗定时器的值（0x0410~0x0411）为零将使看门狗无效。PDI 看门狗超时后，将触发以下操作：

- 设置 ESC DL 状态寄存器 0x0110.1，DL 状态变化映射到 ECAT 帧中并将其发给主站；
- PDI 看门狗超时计数寄存器（0x0443）值增加。

表 3.15 ESC 看门狗相关寄存器

地 址	位	名 称	描 述	复位值
0x0110	1	PDI 看门狗状态	0: PDI 看门狗超时 1: PDI 看门狗在运行或未使能	0
0x0400: 0x0401	0~15	看门狗计时分频率 WD_DIV	设定看门狗计时分频率，例如：默认值 2498 = 100 μs	0x09C2
0x0410: 0x0411	0~15	PDI 看门狗定时器 t _{WD_PDI}	基本看门狗计时单元计数值	0x03e8

续表 3.25

地址	位	名称	描述	复位值
0x0420: 0x0421	0~15	过程数据看门狗定时器 t_{WD_PD}	基本看门狗计时单元计数值	0x03e8
0x0440: 0x0441	0	过程数据看门狗状态	0: 过程数据看门狗超时 1: 过程数据看门狗在运行或未使能	0
	1~15	保留		
0x0442	0~7	过程数据看门狗超时计数	每次当过程数据看门狗超时则计数, 达到 0xFF 时停止, 写操作后清除计数	0
0x0443	0~7	PDI 看门狗超时计数	每次当 PDI 看门狗超时则计数, 达到 0xFF 时停止, 写操作后清除计数	0
0x0804+ N*8	6	SM 看门狗使能	0: 不使能 1: 使能	

3.5 存储同步管理

3.5.1 存储同步管理器概述

ESC 内部过程数据存储区可以用于 EtherCAT 主站与从站应用程序数据的交换, 但必须满足以下要求:

- ① 保证数据一致性, 必须由软件实现协同的数据交换;
- ② 保证数据安全, 必须由软件实现安全机制;
- ③ EtherCAT 主站和应用程序都必须轮询存储器来判断另一端是否完成访问。

ESC 使用了存储同步管理通道 SM (SyncManager) 来保证主站与本地应用数据交换的一致性和安全性, 并在数据状态改变时产生中断来通知双方。SM 通道把存储空间组织为一定大小的缓存区, 由硬件控制对缓存区的访问。缓存区的数量和数据交换方向可配置。SM 由主站配置, 其配置寄存器如表 3.26 所列。SM 配置寄存器从 0x800 开始, 每个通道使用 8 个字节, 包括配置寄存器和状态寄存器。

表 3.16 SM 配置寄存器

偏移地址	位	名称	描述	复位值
+0x0: 0x1	0~16	数据物理起始地址	SM 处理的第一个字节在 ESC 地址空间内的起始地址	0
+0x2: 0x3	0~16	SM 数据长度	分配给 SM 通道的数据长度, 必须大于 1, 否则 SM 将不被激活; 设置为 1 时, 只使能看门狗	0
+0x4	0~7	SM 控制寄存器		0
	0~1	运行模式	00: 3 个缓存区模式 01: 保留 10: 单个缓存区模式 11: 保留	00

续表 3.26

偏移地址	位	名称	描述	复位值
+0x4	2~3	方向	00: 读, ECAT 帧读操作, PDI 写操作	00
	4	ECAT 帧中断请求触发	0: 不使能 1: 使能	0
	5	PDI 中断请求触发	0: 不使能 1: 使能	0
	6	看门狗触发	0: 不使能 1: 使能	
	7	保留		
+0x5	0~7	SM 状态寄存器		
	0	写中断	1: 写操作成功后触发中断 0: 读第一个字节后清除	0
	1	读中断	1: 读操作成功后触发中断 0: 写第一个字节后清除	0
	2	保留		
	3	单缓存区状态	单缓存区模式下, 表示缓存区状态 0: 缓存区空闲 1: 缓存区满	0
	4~5	三个缓存区模式状态	三个缓存区模式下, 表示最后写入的缓冲区 00: 缓存区 1 01: 缓存区 2 10: 缓存区 3 11: 没有写入缓存区	11
	6~7	保留		
+0x6	0~7	ECAT 帧控制 SM 激活		
	0	SM 使能	0: 不使能, 不使用 SM 控制对内存的访问 1: 使能, SM 激活, 控制设置其中的内存访问	0
	1	重复请求	请求重复邮箱数据传输, 主要与 ECAT 帧读邮箱数据一起使用	0
	2~5	保留		
	6	ECAT 帧访问事件锁存	0: 无操作 1: EtherCAT 主站读写一个缓存区后产生锁存事件	
	7	PDI 访问事件锁存	0: 无操作 1: PDI 读写一个缓存区或 PDI 访问缓存区起始地址时产生锁存事件	
+0x7	0~7	PDI 控制 SM		

续表 3.26

偏移地址	位	名称	描述	复位值
	0	使 SM 无效	读和写的含义不同 读: 0: 正常操作, SM 激活 1: SM 无效, 并锁定对内存区的访问 写: 0: 激活 SM 1: 请求 SM 无效, 直到当前正在处理的数据帧结束	0
	1	重复请求应答	与重复请求位相同时, 表示 PDI 对前面设置的重复请求的应答	0
	2~7	保留		0

必须从起始地址开始操作一个缓存区, 否则操作被拒绝。操作起始地址之后, 就可以操作整个缓存区。允许再次操作起始地址, 并且可以分多次操作。操作缓存区的结束地址表示缓存区操作结束, 随后缓存区状态改变, 同时可以产生一个中断信号或看门狗触发脉冲。不允许在一个数据帧内两次操作结束地址。

EtherCAT 定义了两种 SM 运行模式:

(1) **缓存类型** (常用于过程数据通信)

- 使用 3 个缓存区保证可以随时接收和交付最新的数据;
- 经常有一个可写入的空闲缓存区;
- 在第一次写入之后, 经常有一个连续可读的数据缓存区;

(2) **邮箱类型**

- 使用一个缓存区, 支持握手机制;
- 数据溢出保护;
- 只有写入新数据后才可以进行成功的读操作;
- 只有成功读取之后才允许再次写入。

3.5.2 缓存类型数据交换

缓存模式使用三个缓存区, 允许 EtherCAT 主站和从站控制微处理器双方在任何时候访问数据交换缓存区。数据接收方可以随时得到一致的最新数据, 而数据发送方也可以随时更新缓存区的内容。如果写缓存区的速度比读缓存区的速度快, 旧的数据将被覆盖。三个缓存区模式通常用于**周期性过程数据交换**。

在物理上, 三个缓存区由 SM 统一管理, SM 只配置了第一个缓存区的地址范

0x1000 ~ 0x10 FF	缓存区1, 可以直接访问
0x1100 ~ 0x11 FF	缓存区2, 不可以直接访问, 不可以用于其他SM通道
0x1200 ~ 0x12 FF	缓存区3, 不可以直接访问, 不可以用于其他SM通道
0x1300 ...	可用存储空间

图 3.17 SM 通道缓存区分配

注: 所有缓存区由 SM 通道控制, 只有缓存区 1 的地址配置给 SM 通道, 并由主站和本地应用直接访问。

围, 根据 SM 通道的状态, 对第一个缓存区的访问将被重新定向到三个缓存区中的一个。第二和第三个缓存区的地址范围不能被其他 SM 所使用, 例如如图 3.20 所示, 配置了一个 SM 通道, 其起始地址为 0x1000, 长度为 0x100, 则 0x1100~0x12FF 的地址范围不能被直接访问, 而是作为缓存区由 SM 通道来管理。

缓存模式的运行原理如图 3.21 所示。在情况①中, 缓存区 1 正由主站数据帧写入数据, 缓存区 2 空闲, 缓存区 3 由从站微处理器读走数据。主站写缓存区 1 完成后, 缓存区 1 和缓存区 2 交换, 成为图 3.21 中情况②。从站微处理器读缓存区 3 完成后, 缓存区 3 空闲, 并与缓存区 1 交换, 成为图 3.21 中的情况③, 此时, 主站和微处理器又可以分别开始写和读操作。如果 SM 控制寄存器 (0x0804+Nx8) 中使能了 ECAT 帧或 PDI 中断, 那么每次成功的读写操作都将在 SM 状态寄存器 (0x0805+Nx8) 中设置中断事件请求, 并映射到 ECAT 帧中断请求寄存器 (0x0210~0x0211) 和 AL 事件请求寄存器 (0x0220~0x0221) 中, 再由相应的中断屏蔽寄存器决定是否映射到数据帧状态位或触发中断信号。

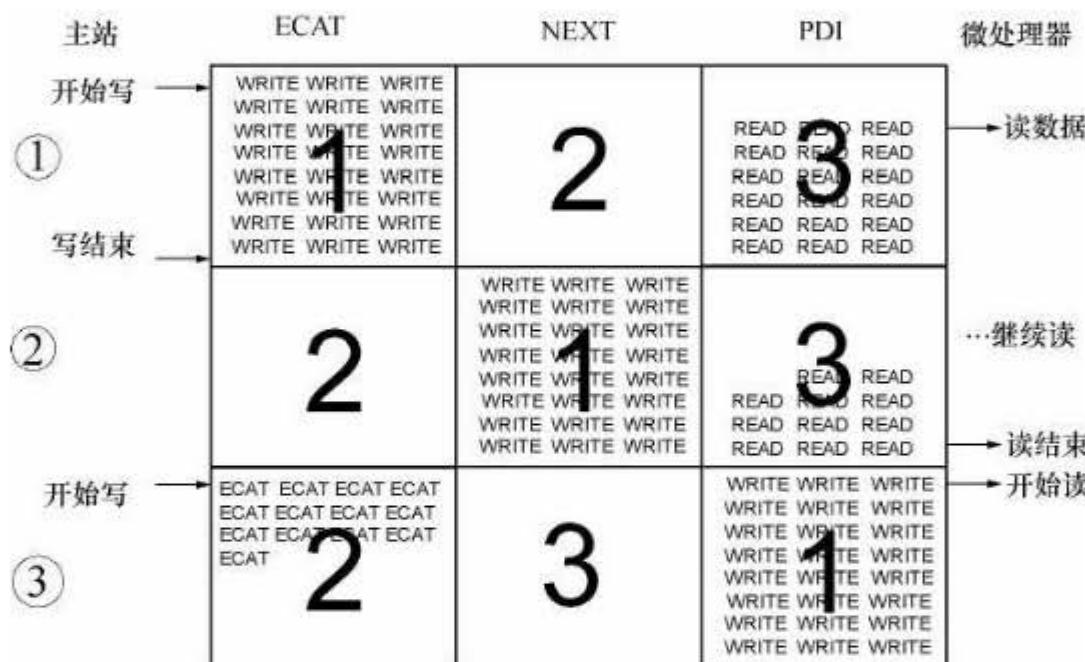


图 3.18 SM 缓存区运行原理

3.5.3 邮箱数据通信机制

邮箱模式使用一个缓存区, 实现了带有握手机制的数据交换, 所以不会丢失数据。只有在一端完成数据操作之后另一端才能访问缓存区。首先, 数据发送方写缓存区, 然后缓存区被锁定为只读, 直到数据接收方读走数据。随后, 发送方再次写操作缓存区, 同时缓存区对接收方锁定。邮箱模式通常用于应用层非周期性数据交换, 分配的这一个缓存区也称为邮箱。邮箱模式只允许以轮流方式读和写操作, 实现完整的数据交换。

只有 ESC 接收数据帧 FCS 正确时 SM 通道的数据状态才会改变, 这样, 在数据帧结束之后缓存区状态立刻变化。

邮箱数据通信使用两个存储同步管理器通道。通常, 主站到从站通信使用 SM0, 从站到主站通信使用 SM1, 它们被配置成为一个缓存区方式, 使用握手来避免数据溢出。

(1) 主站写邮箱操作

主站要发送非周期性数据给从站时，发送 ECAT 帧命令写从站的 SM0 所管理的缓存区地址，邮箱数据通信的子报文格式见“2.5.2 非周期性邮箱数据通信”本节。Ctr 是用于重复检测的顺序编号，每个新的邮箱服务将加 1。数据帧返回主站后，主站检查 ECAT 帧命令的 WKC，如果 WKC 为 1，表示写 SM0 成功，如图 3.22 中的情况①。如果 WKC 仍然为 0，表示 SM0 非空，从站还没有将上次写入的数据读走，主站本次写失败。等待一段时间后再重新发送相同的数据帧，并再次根据返回数据帧的 WKC 判断是否成功，如果从站在此期间读走了缓存区数据，则主站此次写操作成功，返回数据帧子报文的 WKC 等于 1，如图 3.22 中的情况②。

如果写邮箱数据帧丢失，主站在发现接收返回数据帧超时之后，重新发送相同数据帧。从站读取此数据之后，发现其中的计数器 Ctr 与上次数据命令相同，表示为重复的邮箱数据。如图 3.22 中的情况③。

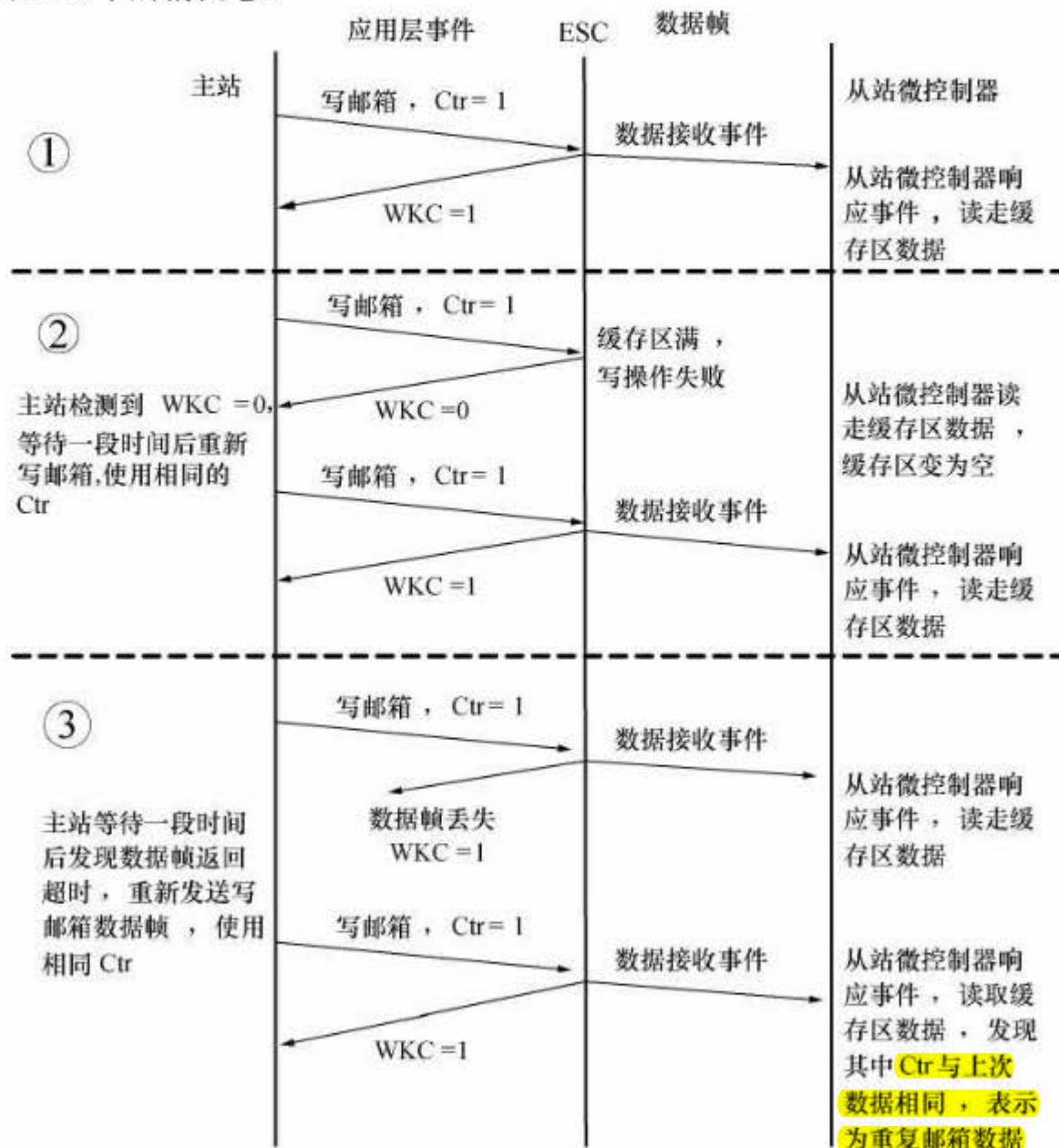


图 3.19 邮箱数据通信机制

(2) 主站读邮箱操作

所有数据交换都是由主站发起的，如果从站有数据要发送给主站，必须先将数据写入

发送邮箱缓存区，然后由主站来读取。主站有两种方法来测定从站是否已经将邮箱数据填入发送数据区。一种方法是将 SM1 配置寄存器中的邮箱状态位（0x80D.3）映射到逻辑地址中，使用 FMMU 周期性地读这一位。使用逻辑寻址可以同时读取多个从站的状态位，这种方法的缺点是每个从站都需要一个 FMMU 单元。另一个方法是简单地轮询 SM1 数据区。从站已经将新数据填入数据区后这个读命令的工作计数器 WKC 将加 1。

读邮箱操作可能会出现错误，主站需要检查从站邮箱命令应答报文中的工作计数器 WKC。如果工作计数器没有增加（通常由于从站没有完成上一个读命令）或在限定的时间内没有响应，主站必须翻转 SM0 控制寄存器中的重复请求位（0x0806.1）。从站检测到翻转位之后，将上次的数据再次写入 SM1 数据区，并翻转 SM1 配置寄存器中 PDI 控制字节中的重发应答位（0x80E.1）。主站读到 SM1 翻转位后，再次发起读命令。主站读邮箱操作的整个过程如图 3.23 所示。

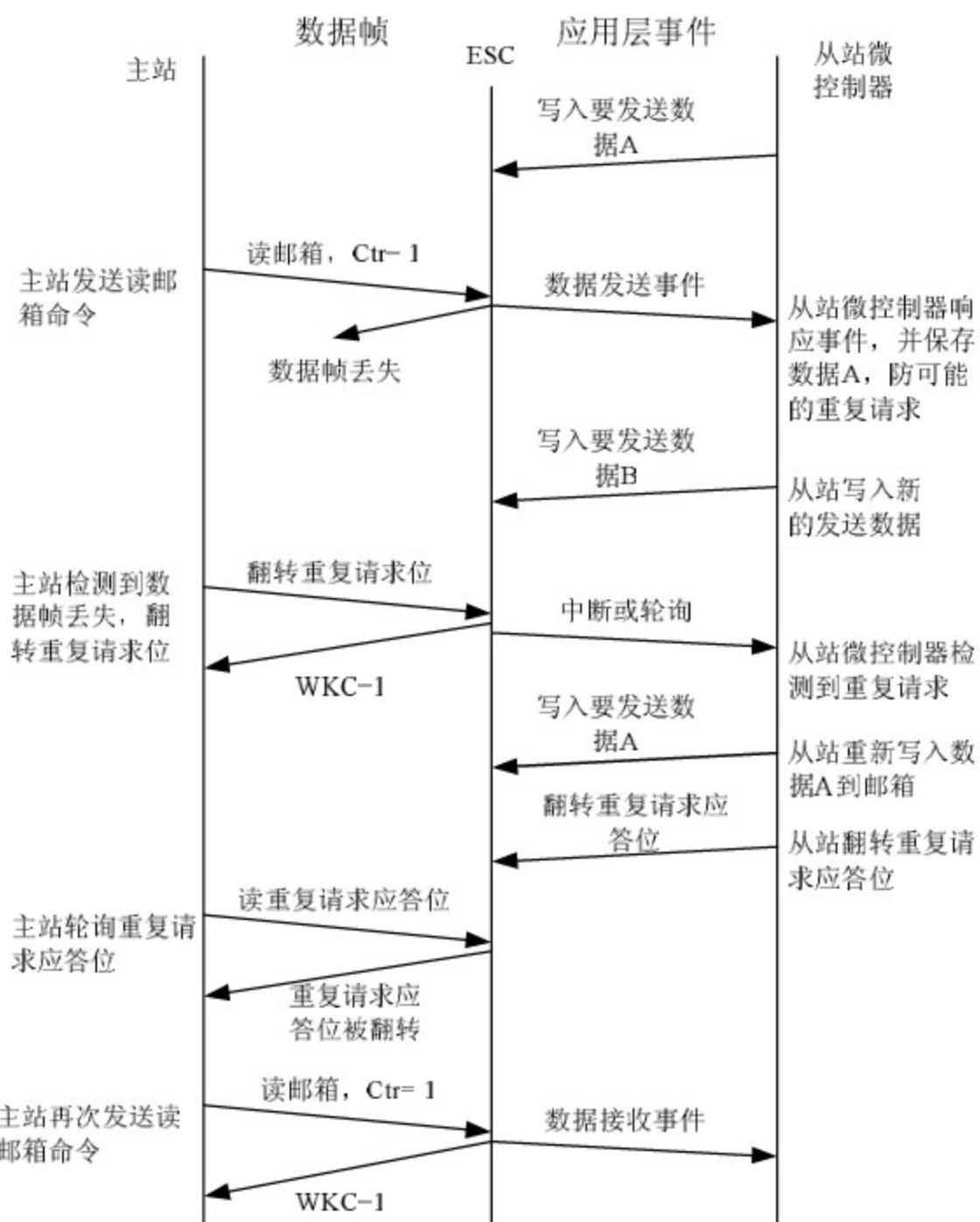


图 3.20 读邮箱数据错误处理

3.6 从站信息接口

ESC 使用 EEPROM 来存储所需要的设备相关信息，称为**从站信息接口 SII(Slave Information Interface)**。EEPROM 的容量为 1 Kbit~4 Mbit，取决于 ESC 规格。EEPROM 内容结构如图 3.24 所示。EEPROM 使用字地址，字 0~63 是必须的基本信息，其各部分描述如下：

- ① ESC 寄存器配置区（字 0~7）由 ESC 在上电或复位后自动读取后装入相应寄存器，并检查校验和；
- ② 产品标识区（字 8~15）包括厂商标识、产品码、版本号和序列号等；
- ③ 硬件延时（字 16~19）包括端口延时和处理延时等信息；
- ④ 引导状态下邮箱配置（字 20~23）；
- ⑤ 标准邮箱通信 SM 配置（字 24~27）。



图 3.21 EEPROM 数据布局

3.6.1 EEPROM 内容

ESC 配置区数据如表 3.27 所列。

表 3.17 ESC 配置数据内容

字地址	参数名	描 述
0	PDI 控制	PDI 控制寄存器初始值 (0x0140:0x0141)
1	PDI 配置	PDI 配置寄存器初始值 (0x0150:0x0151)
2	SYNC 信号脉冲宽度	SYNC 信号脉宽寄存器初始值 (0x0982:0x0983)

续表 3.27

字地址	参数名	描述
3	扩展 PDI 配置	扩展 PDI 配置寄存器初始值 (0x0152:0x0153)
4	站点别名	站点别名配置寄存器初始值 (0x0012:0x0013)
5,6	保留	保留, 应为 0
7	校验和	字 0~6 的校验和

EEPROM 中的分类附加信息包含了可选的从站信息，有两种类型的数据：标准类型和制造商定义类型。所有分类数据都使用相同的数据结构，包括一个字的数据类型、一个字的数据长度和数据内容，如图 3.25 所示。其标准的分类数据类型如表 3.28 所列。

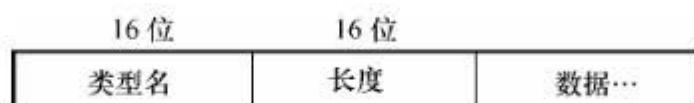


图 3.22 EEPROM 分类数据结构图

表 3.18 分类数据类型

类型名	数 值	描 述
STRINGS	10	文本字符串信息
General	30	设备信息
FMMU	40	FMMU 使用信息
SyncM	41	存储同步管理器运行模式
TXPDO	50	TxDPO 描述
RXPDO	51	RxDPO 描述
DC	60	分布式时钟描述
End	0xffff	分类数据结束

3.6.2 EEPROM 访问控制

ESC 具有读写 EEPROM 的功能，主站或 PDI 通过读写 ESC 的 EEPROM 控制寄存器来读写 EEPROM，在复位状态下由主站控制 EEPROM 的操作，之后可以移交给 PDI 控制。EEPROM 控制寄存器功能如表 3.29 所列。

表 3.19 EEPROM 控制寄存器

地 址	位	名 称	描 述	复位值
0x0500	0	EEPROM 访问分配	0: ECAT 帧 1: PDI	0
	1	强制 PDI 操作释放	0: 不改变 0x0501.0 1: 复位 0x0501.0 为 0	0
	2~7	保留		0
0x0501	0	PDI 操作	0: PDI 释放 EEPROM 操作 1: PDI 正在操作 EEPROM	0
	1~7	保留		0
0x0502	0~15	EEPROM 控制和状态寄存器		
0x0503	0	ECAT 帧写使能	0: 写请求无效 1: 使能写请求	0
	1~5	保留		
	6	支持读字节数	0: 4 个字节 1: 8 个字节	ET1100: 1 ET1200: 1 其他: 0
	7	EEPROM 地址范围	0: 1 个地址字节 (1KBit-16KBit) 1: 2 个地址字节 (32KBit-4MBit)	芯片配置引脚
	8	读命令位	读写操作时含义不同。 写 0: 无操作 1: 开始读操作 读 0: 无读操作 1: 读操作进行中	0
	9	写命令位	读写操作时含义不同。 写 0: 无操作 1: 开始写操作 读 0: 无写操作 1: 写操作进行中	0
	10	重载命令位	读写操作时含义不同。 写 0: 无操作 1: 开始重载操作 读 0: 无重载操作 1: 重载操作进行中	0
	11	ESC 配置区校验	0: 校验和正确 1: 校验和错误	0

续表 3.29

地 址	位	名 称	描 述	复位值
	12	器 件 信 息 校 验	0: 器件信息正确 1: 从 EEPROM 装载器件信息错误	0
	13	命 令 应 答	0: 无错误 1: EEPROM 无应答, 或命令无效	0
	14	写 使 能 错 误	0: 无错误 1: 请求写命令时无写使能	0
	15	繁 忙 位	0: EEPROM 接口空闲 1: EEPROM 接口繁忙	0
0x0504: 0x0507	0~32	EPPROM 地址	请求操作的 EEPROM 地址, 以字为单位	0
0x508: 0x50F	0~15	EPPROM 数据	将写入 EEPROM 的数据, 或从 EEPROM 读到数据, 低位字	0
	16~63	EPPROM 数据	从 EEPROM 读到数据, 高位字, 一次读 4 个字节时只有 16~31 有效	0

存器 0x0500 和 0x0501 分配 EEPROM 的访问控制权。如果 $0x0500.0 = 0$ 、 $0x0501.0 = 0$, 则由主站控制 EEPROM 访问接口, 这也是 ESC 的默认状态; 否则, 由 PDI 控制 EEPROM。双方在使用 EEPROM 之前需要检查访问权限。EEPROM 访问权限的移交有主动放弃和被动剥夺两种形式。双方在访问完成后可以主动放弃控制权, 而主站可以在 PDI 没有释放控制权时强制获取操作控制, 操作如下:

- (1) 主站操作 EEPROM 结束后, 主动写 $0x0500.0 = 1$, 将 EEPROM 接口移交给 PDI;
- (2) 如果 PDI 想要操作 EEPROM, 则写 $0x0501.0 = 1$, 接管 EEPROM 控制;
- (3) PDI 完成 EEPROM 操作后, 写 $0x0501.0 = 0$, 释放 EEPROM 操作;
- (4) 主站写 $0x0500.0 = 0$, 接管 EEPROM 控制权;
- (5) 如果 PDI 未主动释放 EEPROM 控制, 主站可以写 $0x0500.1 = 1$, 强制清除 $0x0501.0$, 从 PDI 夺取 EEPROM 控制;

EEPROM 接口支持 3 种操作命令: 写一个 EEPROM 地址、从 EEPROM 读或从 EEPROM 重载 ESC 配置。需要按照以下步骤执行读/写 EEPROM 的操作:

- (1) 检查 EEPROM 是否空闲 ($0x0502.15$ 是否为 0), 如果不空闲, 则必须等待, 直到空闲;
- (2) 检查 EEPROM 是否有错误 ($0x0502.13$ 是否为 0 或 $0x0502.14$ 是否为 0), 如果有错误, 则写 $0x0502.[10:8] = [000]$ 清除错误。
- (3) 写 EEPROM 字地址到 EEPROM 地址寄存器。
- (4) 如果要执行写操作, 首先将要写入的数据写入 EEPROM 数据寄存器 $0x0508\sim0x0509$ 。

(5) 写控制寄存器启动命令执行:

- 读操作，写 $0x500.8 = 1$ 。
- 写操作，写 $0x500.0=1$ 和 $0x500.9=1$ ，这两位必须由一个数据帧写完成。 $0x500.0$ 为写使能位，可以实现写保护机制；它对同一数据帧中的 EEPROM 命令有效，并随后自动清除；对于 PDI 访问控制不需要写这一位。
- 重载操作，写 $0x500.10=1$ 。

(6) 主站发起读写操作是，在数据帧结束符 EOF (End Of Frame) 之后开始执行的；PDI 发起的操作则马上被执行。

- (7) 等待 EEPROM 繁忙位清除 ($0x0502.15$ 是否为 0)；
- (8) 检查 EEPROM 错误位。如果 EEPROM 应答丢失，可以重新发起命令（回到第(5)步）。在重试之前等待一段时间，使 EEPROM 有足够时间保存内部数据。

(9) 获得执行结果：

- 读操作，读到的数据在 EEPROM 数据寄存器 $0x0508\sim0x050F$ 中，数据长度可以是 2 或 4 个字，取决于 $0x0502.6$ ；
- 重载操作，ESC 配置被重新写入相应的寄存器。

3.6.3 EEPROM 操作错误处理

EEPROM 接口操作错误由 EEPROM 控制/状态寄存器 $0x0502\sim0x0503$ 指示，可能发生的错误如表 3.30 所列。

表 3.20 EEPROM 接口错误

Bit	名 字	描 述
11	校验和错误	ESC 配置区域校验和错误，使用 EEPROM 初始化的寄存器保持原值 原因：CRC 错误 解决方法：检查 CRC
12	设备信息错误	ESC 配置没有被装载 原因：校验和错误、应答错误或 EEPROM 丢失 解决方法：检查其他错误位
13	应答/命令错误	无应答或命令无效 原因：① EEPROM 芯片无应答信号 ② 发起了无效的命令 解决方法：① 重试访问 ② 使用有效的命令
14	写使能错误	主站在没有写使能的情况下执行了写操作 原因：主站在写使能位无效时发起了写命令 解决方法：在写命令的同一个数据帧中设置写使能位

ESC 在上电或复位后读取 EEPROM 中的配置数据，如果发生错误，则重试读取。连

续两次读取失败后，设置设备信息错误位，此时 ESC 数据链路状态寄存器中 PDI 允许运行位（0x0110.0）保持无效。发生错误时，所有由 ESC 配置区初始化的寄存器保持其原值，ESC 过程数据存储区也不可访问，直到成功装载 ESC 配置数据。

EEPROM 芯片无应答错误是一个常见的问题，更容易在 PDI 操作 EEPROM 时发生。连续写 EEPROM 时产生无应答错误原因如下：

- ① 主站或 PDI 发起第一个写命令；
- ② ESC 将写入数据传送给 EEPROM 芯片；
- ③ EEPROM 芯片内部将输入缓存区中数据传送到存储区，有可能需要几毫秒；
- ④ 主站或 PDI 发起第二个写命令；
- ⑤ ESC 将写入数据传送给 EEPROM 芯片，EEPROM 芯片不应答任何访问，直到上次内部数据传送完成；
- ⑥ ESC 设置应答/命令错误位；
- ⑦ EEPROM 芯片完成内部数据传送；
- ⑧ ESC 重新发起第二个命令，命令被应答并成功执行。

3.7 分布时钟操作

分布时钟由主站在数据链路的初始化阶段进行初始化、配置和启动运行。在运行阶段，主站也需要维护分布时钟的运行，补偿时钟漂移。在从站端，分布时钟由 ESC 芯片实现，为从站控制微处理器提供同步的中断信号和时钟信息。时钟信息也可以用于记录锁存输入信号的时刻。

3.7.1 分布式时钟信号

1. 同步信号

分布时钟控制单元可以产生两个同步信号 SYNC0 和 SYNC1，用于给应用程序提供中断或直接触发的输出数据更新。同步信号的控制相关寄存器如表 3.31 所列。

表 3.21 同步信号相关寄存器

地 址	位	名 称	描 述	复位值
0x0980	0	SYNC 输出单元控制	0: 主站控制 1: PDI 控制	0
	1~3	保留		0
	4	锁存输入单元 0 控制	0: 主站控制 1: PDI 控制	0
	5	锁存输入单元 1 控制	0: 主站控制 1: PDI 控制	0
	6~7	保留		

续表 3.31

地 址	位	名 称	描 述	复位值
0x0981	0	激活周期运行	0: 无效 1: 如果 SYNC0 周期时间=0, 只产生一个 SYNC 脉冲	0
	1	激活 SYNC0	0: 无效 1: 产生 SYNC0 脉冲	0
	2	激活 SYNC1	0: 无效 1: 产生 SYNC1 脉冲	0
	3~7	保留		0
0x0982:	0~15	SYNC 脉冲宽度	SYNC 信号宽度, 以 10 ns 为单位	EEPROM 地址 0x2
0x0983			0: 应 答 模 式, SYNC 信号由读取 SYNC0/SYNC1 状态寄存器清除	
0x098E	0	SYNC0 状态	应答模式时读此寄存器将清除 SYNC0 信号	0
	1~7	保留		0
0x098F	0	SYNC1 状态	应答模式时读此寄存器将清除 SYNC1 信号	0
	1~7	保留		0
0x0990:	0~63	周期运行开始时间	写: 周期性运行开始时间, 以 ns 为单位 读: 下一个 SYNC0 脉冲信号时间, 以 ns 为单位	0
0x0997				
0x0998:	0~63	SYNC1 时间	下一个 SYNC1 时间, 以 ns 为单位	0
0x099F				
0x09A0:	0~31	SYNC0 周期时间	两个连续 SYNC0 脉冲之间的时间, 以 ns 为单位,	0
0x09A3			0: 单脉冲模式, 只产生一个 SYNC0 脉冲	
0x09A4:	0~31	SYNC1 周期时间	SYNC1 脉冲和 SYNC0 脉冲之间的时间, 以 ns 为单位	0
0x09A7				

同步信号的宽度由脉宽寄存器 0x0982: 0x0983 设定, SYNC0 信号周期时间由 SYNC0 周期时间寄存器 (0x09A0~0x09A3) 设置。在同步单元被激活, SYNC0/1 信号输出被使能后, 同步单元等待开始时间到达后产生第一个 SYNC0 脉冲。SYNC 信号的刷新频率是 100 MHz (10 ns 刷新周期)。SYNC 信号与系统时间之间的抖动为 12ns。脉宽寄存器和 SYNC0 周期时间共同决定了 SYNC0 信号的运行模式, 如表 3.32 所列。

表 3.22 同步信号运行模式选择

SYNC0/1 信号脉宽寄存器 (0x0982: 0x0983)	SYNC0 周期时间 (0x09A0: 0x09A3)	
	>0	=0
>0	周期	单次
=0	周期性应答	单次应答

同步信号可以有 4 种模式，如图 3.26 所示。应答模式通常用于产生中断，中断信号必须由微处理器响应后才能恢复。这四种运行模式功能如下。

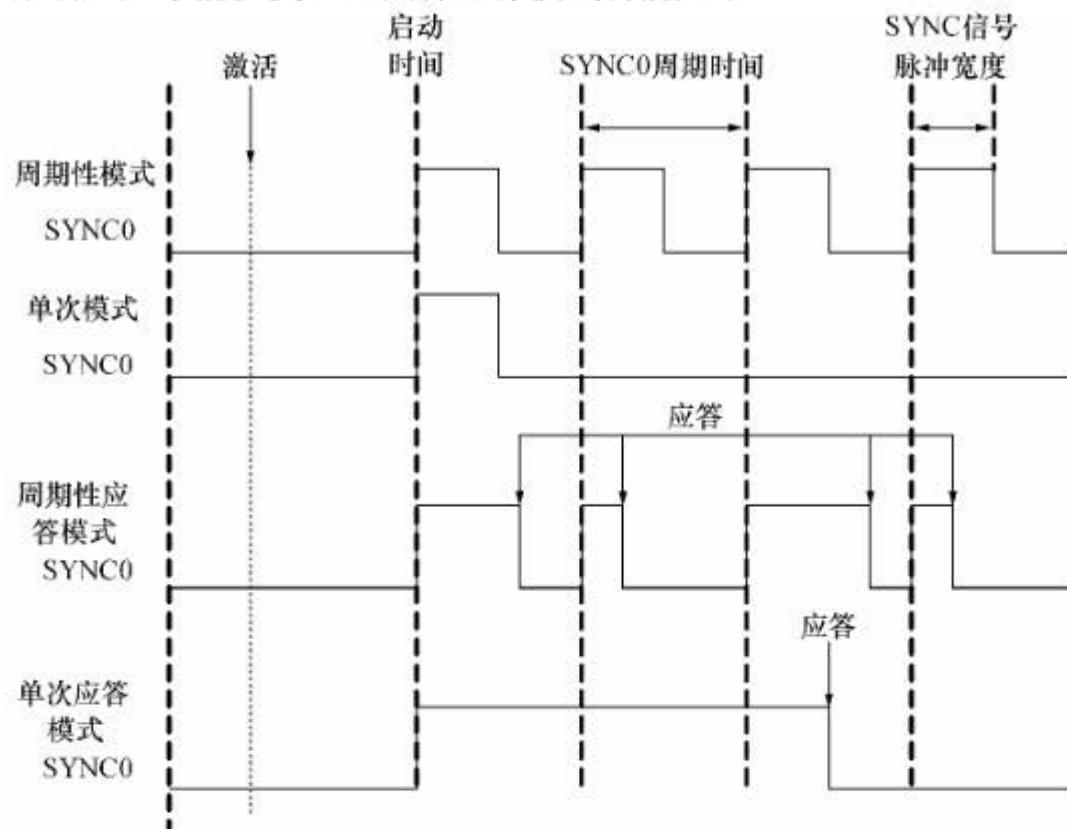


图 3.23 同步信号产生模式

(1) 周期性模式

在周期性模式下，分布时钟控制单元在启动操作后产生等时的同步信号。在终止操作后停止运行。周期时间由 SYNC0/1 周期时间寄存器决定。SYNC 信号的脉冲宽度必须大于 0，如果脉冲宽度大于周期时间，则 SYNC 信号将在启动后总保持有效。

(2) 单次模式

单次模式下（SYNC0 周期时间设为 0），在启动时间到达后只产生一个同步信号脉冲。在重新写入开始时间并重新启动周期单元后可以产生下一个脉冲。

(3) 周期性应答模式

周期性应答模式的典型应用是产生等时中断。通过设置 SYNC0 信号脉冲宽度（寄存器 0x0982~0x0983）为 0 选择应答模式。SYNC 信号在获得应答之前保持有效，由微处理器读 SYNC0 或 SYNC1 状态寄存器（0x098E, 0x098F）产生应答。第一个脉冲在启动时间到达后产生，之后的脉冲在下一个 SYNC0/1 事件发生时产生。

(4) 单次应答模式

单次应答模式下，启动时间到达时只产生一个脉冲。在读 SYNC0/1 状态寄存器产生应答之前脉冲保持有效。重新写入开始时间并重新启动控制单元后可以产生下一个脉冲。

第二个同步信号 SYNC1 依赖于 SYNC0，可以比 SYNC0 延迟一个预定义的量。延迟量由 SYNC1 周期时间寄存器（0x09A4: 0x09A7）设置。SYNC1 与 SYNC0 并非一一对应，**SYNC1 总以其后的下一个 SYNC0 信号为参照基准**，如图 3.27 所示。

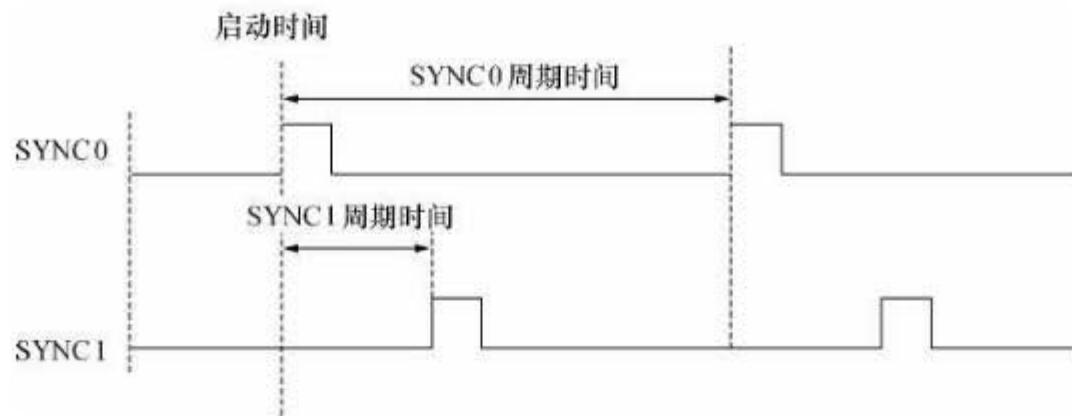


图 3.24 SYNC1 信号产生示例

同步信号的产生需要以下初始化过程：

- (1) 设备上电，自动从 EEPROM 装载默认值（如表 3.27 所示）；
 - PDI 控制寄存器 0x0140.10=1，使能 DC 同步信号输入单元；
 - 同步/锁存 PDI 配置寄存器 0x0151，使 SYNC0/1 使用适当的输出驱动模式；
 - 脉冲宽度寄存器 0x0982 : 0x0983。
- (2) 设置寄存器 0x0980，分配同步单元给 ECAT 帧或 PDI 控制，决定后续设置参数操作由 ECAT 帧或 PDI 执行，默认值为由 ECAT 帧执行；
- (3) 设置 SYNC0 和 SYNC1 信号周期时间；
- (4) 设置周期性运行启动时间，启动时间必须是在周期性运行激活时刻之后，否则必须等计数器溢出后才开始周期性运行；
- (5) 设置寄存器 0x0981.0=1 激活周期性运行，并使能 SYNC0/1 信号输出（设 0x981[2:1]=0x03），同步单元在周期性运行启动时间到达后开始产生 SYNC0 脉冲。

2. 同步锁存信号

DC 锁存单元可以为两个外部事件信号保存时间标记，外部事件信号称为：Latch0 和 Latch1。上升沿和下降沿的时间标记都被记录。另外，有些从站也可以记录存储同步管理器事件时间标记。

锁存信号的采样率为 100 MHz，相应时间标记的内部抖动为 11 ns。锁存信号的状态可以从锁存状态寄存器读取。DC 锁存单元支持两种模式：单事件或连续模式，由 Latch0/1 控制寄存器（0x09A8: 0x09A9）配置。

(1) 单事件模式

在单事件模式下，只有锁存信号的第一个上升沿和第一个下降沿的时间标记被记录。锁存状态寄存器（0x09AE: 0x09AF）包含已经发生的事件的信息。锁存时间寄存器（0x9B0~0x9CF）包含时间标记。

每个事件都通过读相应的锁存时间寄存器来应答。锁存时间寄存器被读取后，锁存单元等待下一个事件发生。在单事件模式下，锁存事件也映射到 AL 事件请求寄存器中。

(2) 连续模式

在连续模式下，每个事件的时间都被保存在锁存时间寄存器中（0x9B0~0x9CF）。每次读取都读到最近发生的事件的时间标记。在连续模式下，锁存状态寄存器（0x09AE:

0x09AF) 不反映锁存事件的状态。

(1) 锁存主站读写

有些从站支持锁存主站读写区事件时间来调试存储同步管理器的操作。如果 SM 配置正确，最近的事件可以从 SM 事件时间标记寄存器 (0x09F0:0x09FF) 读取。

分布式时钟中的同步信号单元和两个锁存信号单元可以由 ECAT 帧控制或本地微处理器 (PDI) 控制，主站通过写周期单元控制寄存器 0x0980 分配控制权。通过 PDI 控制，微处理器可以根据自己的需求配置分布式时钟，例如设定周期性的中断。

表 3.33 列出了用于锁存信号时间标记的寄存器。

表 3.23 锁存信号相关寄存器

地 址	位	名 称	描 述	复位值
0x0140	11:10	PDI 控制	使能/终止 DC 单元 (低功耗)	EEPROM
0x0151		同步/锁存 PDI 配置	配置同步/锁存信号引脚	EEPROM
0x09A8	0~7	Latch0 控制		
	0	Latch0 上升沿控制	0: 连续锁存有效 1: 单次事件模式，只有第一个事件有效	0
	1	Latch0 下降沿控制	0: 连续锁存有效 1: 单次事件模式，只有第一个事件有效	0
	2~7	保留		0
	0~7	Latch1 控制		
0x09A9	0	Latch1 上升沿控制	0: 连续锁存有效 1: 单次事件模式，只有第一个事件有效	0
	1	Latch1 下降沿控制	0: 连续锁存有效 1: 单次事件模式，只有第一个事件有效	0
	2~7	保留		0
	0~7	Latch0 状态		
0x09AE	0	Latch0 上升沿状态	发生 Latch0 上升沿事件，单次模式有效，否则为 0， 读 Latch0 上升沿时间寄存器清除	0
	1	Latch0 下降沿状态	发生 Latch0 下降沿事件，单次模式有效，否则为 0， 读 Latch0 下降沿时间寄存器清除	0
	2	Latch0 引脚状态	Latch0 引脚的状态	0
	3~7	保留		0
	0~7	Latch1 状态		
0x09AF	0	Latch1 上升沿状态	发生 Latch1 上升沿事件，单次模式有效，否则为 0， 读 Latch1 上升沿时间寄存器清除	0
	1	Latch1 下降沿状态	发生 Latch1 下降沿事件，单次模式有效，否则为 0， 读 Latch1 下降沿时间寄存器清除	0
	2	Latch1 引脚状态	Latch1 引脚的状态	0
	3~7	保留		0

续表 3.33

地 址	位	名 称	描 述	复位值
0x09B0: 0x09B7	0~63	Latch0 上升沿时间	Latch0 信号上升沿捕获时系统时间	0
0x09B8: 0x09BF	0~63	Latch0 下降沿时间	Latch0 信号下降沿捕获时系统时间	0
0x09C0: 0x09C7	0~63	Latch1 上升沿时间	Latch1 信号上升沿捕获时系统时间	0
0x09C8: 0x09CF	0~63	Latch1 下降沿时间	Latch1 信号下降沿捕获时系统时间	0
0x09F0: 0x09F3	0~31	主站读写事件时间	捕获到至少一个 SM 发生 ECAT 帧事件时的系统时间	0
0x09F8: 0x09FB	0~31	PDI 缓存区开始事件 时间	捕获到至少一个 SM 发生 PDI 缓存区开始事件时的系 统时间	0
0x09FC: 0x09FF	0~31	PDI 缓存区改变事件 时间	捕获到至少一个 SM 发生 PDI 缓存区改变事件时的系 统时间	0

3.7.2 分布时钟的初始化

分布时钟的初始化过程原理参考 2.4 节。其中的功能都通过读/写寄存器来实现。分布时钟初始化相关寄存器如表 3.34 所列。

表 3.24 分布时钟相关寄存器

地 址	位	名 称	描 述	复位值
0x0900: 0x0903	0: 31	端口 0 接收时刻	读和写功能不同。 写: 写 0x0900, 各端口锁存数据帧第一个前导位到达时的本地时间 读: 读锁存的数据帧第一个前导位到达端口 0 时的本地系统时间	无
0x0904: 0x0907	0: 31	端口 1 接收时刻	读: 读锁存的数据帧第一个前导位到达端口 1 时的本地时间	无
0x0908: 0x090B	0: 31	端口 2 接收时刻	读: 读锁存的数据帧第一个前导位到达端口 2 时的本地时间	无
0x090C: 0x090F	0: 31	端口 3 接收时刻	读: 读锁存的数据帧第一个前导位到达端口 3 时的本地时间	无
0x0910: 0x0917	0: 63	本地系统 时间	每个数据帧第一个前导位到达时锁存的本地系统时间副本 写: 比较写入值和本地系统时间副本, 结果作为时间控制环的输入 读: 获得本地系统时间	0
0x0918: 0x091F	0: 63	数据帧处理单 元接收时间	读: 读锁存的数据帧第一个前导位到达数据帧处理单元时的本地时 间	无

续表 3.34

地 址	位	名 称	描 述	复位值
0x0920: 0x0927	0: 63	时间偏移	本地时间和系统时间的偏差	0
0x0928: 0x092B	0~31	传输延时	参考时钟 ESC 和当前 ESC 之间的传输延时	0
0x092C: 0x092F	0~30 31	系统时间差 符号	本地系统时间副本与参考时钟系统时间值之差 0: 本地系统时间大于或等于参考时钟系统时间 1: 本地系统时间小于参考时钟系统时间	0
0x0930: 0x0931	0~14 15		调节本地系统时间的带宽	0x1000
0x0932: 0x0933	0~15	偏差	本地时钟周期和参考时钟周期偏差	0
0x0934	0~3 4~7	过滤深度 保留	系统时间偏差计算的平均次数	4
0x0935	0~3	过滤深度	时钟周期偏差计算的平均次数	12

从初始化阶段到预运行阶段，在发送从站初始化命令之前，必须执行以下操作：

① 主站读所有从站特征信息寄存器 0x0008~0x0009（参见表 3.4），根据 bit2 的值得知哪些从站支持分布时钟。由于此时处在初始化阶段，所以使用顺序寻址命令 APRD 操作并获得从站 DC 特征信息：

- Bit 2 = 0 : 支持分布式时钟；
- Bit 2 = 1 : 不支持分布式时钟；
- Bit 3 = 0 : 支持 64 位分布式时钟；
- Bit 3 = 1 : 支持 32 位分布式时钟。

② 主站读数据链路状态寄存器 0x0110~0x0111，根据其中的端口状态判断正被使用的端口，获得网段拓扑结构。由表 3.19 可知，如果端口被打开，且建立了通信，表示此端口正被使用。各端口相应的状态位如表 3.35 所列。根据各个从站端口通信状态可以获得准确的网络拓扑结构。

表 3.25 端口链路状态判断位

端 口	打 开 标 志	建 立 通 信 标 志
0	bit8 = 0	bit9 = 1
1	bit10 = 0	bit11 = 1
2	bit12 = 0	bit13 = 1
3	bit14 = 0	bit15 = 1

③ 主站发送一个广播写命令 BWR，写所有从站端口 0 的接收时间寄存器 0x0900，将所有从站捕捉数据帧第一个前导位到达每个端口的本地时间保存到寄存器 0x0900~0x090F，每个端口使用 4 个字节。

④ 主站分别读取各个从站以太网帧到达时刻 0x900~0x90F，根据第②步得到的信息来决定哪些端口正被使用；假设只使用端口 0 和端口 1，则根据图 2.19 可知，寄存器和其中的变量有以下对应关系：

- 参考时钟从站寄存器 0x0900~0x0903 中保存了数据帧到达端口 0 时的参考时钟 t_{sys_ref} 时刻 T_1 ；
- 参考时钟从站寄存器 0x0904~0x0907 中保存了数据帧返回时到达端口 1 时的参考时钟 t_{sys_ref} 时刻 T_4 ；
- 从站 n 的寄存器 0x0900~0x0903 中保存了数据帧到达从站 n 端口 0 时本地时钟 $t_{local}(n)$ 的时刻 $T_2(n)$ ；
- 从站 n 的寄存器 0x0904~0x0907 保存了数据帧返回从站 n 端口 1 时本地时钟 $t_{local}(n)$ 的时刻 $T_3(n)$ 。

⑤ 计算传输延时和初始偏移量

主站根据公式（2-4）计算每个从站与参考时钟从站之间的传输延时 $T_{delay}(n)$ ，主站根据公式（2-3）计算从时钟和参考时钟之间的初始偏移量 $T_{offset}(n)$ 。

⑥ 主站使用 APWR 命令将步骤⑤计算得到的 $T_{delay}(n)$ 写入每个从站的传输延时寄存器 0x928~0x92B。

⑦ 主站使用 APWR 命令将初始偏移量 $T_{offset}(n)$ 写入每个从站的初始时间偏差寄存器 0x0920~0x0927。

⑧ 主站使用 ARMW 命令读参考时钟的系统时间寄存器 0x0910~0x0917，然后将读取结果写入到后续所有从站的本地系统时间寄存器 0x0910~0x0917；各个从站根据图 2.20 描述的原理初始化各自的本地时钟。

这一步必须重复多次，可以读取系统时间差寄存器（0x092C~0x092F）来判断时钟同步性是否已达到需求。如果主站时钟也需要同步，主站也可以根据接收到的时间来调整自己的时钟。

⑨ 初始化结束，开始发送周期性数据帧。主站通过读参考时钟 ESC 的系统时间保持与系统时间同步。在运行模式下，也可以经常重复步骤③、④、⑤和⑥，随时修正传输延时的值。

3.7.3 同步信号的配置

系统的时钟同步以后，将为各个从站提供同步的中断信号和精确的时间标记。在预运行阶段，使用以下命令来配置每个从站的同步信号。

(1) 设定同步信号周期时间

主站使用 FPWR 命令写每个从站的 SYNC 信号周期时间寄存器 0x9A0~0x9A7，其中：

- 字节 0x9A0~0x9A3 为 SYNC0 信号的周期时间；
- 字节 0x9A4~0x9A7 为 SYNC1 信号和 SYNC0 信号的间隔时间。

(2) 设置同步信号启动时间

主站使用 FPWR 命令写每个从站的同步信号启动时间寄存器 x990~0x998。

(3) 激活同步信号

主站使用 FPWR 命令写每个从站的同步信号控制寄存器 0x981，其中，

- bit0 = 1：激活周期性运行；
- bit1 = 1：使能 SYNC0 信号；
- bit2 = 1：使能 SYNC1 信号。

第 4 章 EtherCAT 硬件设计

EtherCAT 主站使用标准的以太网设备，能够发送和接收符合 IEEE802.3 标准以太网数据帧的设备都可以作为 EtherCAT 主站。在实际应用中，可以使用基于 PC 计算机或嵌入式计算机的主站，对其硬件设计没有特殊要求。

EtherCAT 从站使用专用 ESC 芯片，需要设计专门的从站硬件。本章给出使用 8 位并行微处理器总线接口的从站硬件和直接 IO 控制的从站硬件设计实例。

4. 1 EtherCAT 从站 PHY 器件选择

ET1100 芯片只支持 MII 接口的以太网物理层 PHY 器件。有些 ESC 器件也支持 RMII (Reduced MII) 接口。但是由于 RMII 接口 PHY 使用发送 FIFO 缓存区，增加了 EtherCAT 从站的转发延时和抖动，所以不推荐使用 RMII 接口是不推荐的。ET1100 的 MII 接口经过优化设计，为降低处理和转发延时，对 PHY 器件有一些额外要求，可参见“3.2.2 物理通信端口”的节。大多数流行的以太网 PHY 都能满足这些要求。另外，为了获得更好的性能，希望 PHY 满足以下条件：

- ① PHY 检测链接丢失的响应时间小于 15 μs，以满足冗余功能要求；
- ② 接收和发送延时稳定；
- ③ 如果标准的最大线缆长度为 100 m，PHY 支持的最大线缆长度应大于 120 m，以保证安全极限；
- ④ ET1100 的 PHY 管理接口 MI (Management Interface) 的时钟引脚也用作配置输入引脚，所以不应固定连接上拉或下拉电阻；
- ⑤ 最好具有波特率和全双工的自动协商功能；
- ⑥ 低功耗；
- ⑦ 3.3 V 单供电电压；
- ⑧ 使用 25 MHz 时钟源；
- ⑨ 支持工业级的使用温度范围。

为了支持用户从站硬件的设计，BECKHOFF 公司给出了一些满足 ET1100 要求的 PHY 器件（如表 4.1 所列）和一些不满足要求的器件（如表 4.2 所列）。

表 4.1 ET1100 兼容的 PHY 器件列表

制造商/器件	物理地址	物理地址偏移	链接丢失响应时间	备注
Broadcom 公司				
BCM5221	0~31	0	1.3 μs	没有经过硬件测试，依据数据手册或厂商提供数据，要求使用石英振荡器，不能使用 CLK25Out，以避免级联的 PLL
BCM5222	0~31	0	1.3 μs	
BCM5241	0~7,8,16,24	0	1.3 μs	
Micrel 公司				
KS8001L	1~31	16		PHY 地址 0 为广播地址
KS8721B KS8721BT KS8721BL KS8721SL KS8721CL	0~31	0	6 μs	KS8721BT 和 KS8721BL 经过硬件测试，MDC 具有内部上拉
National 半导体公司				
DP83640	1~31	16	250 μs	PHY 地址 0 表示隔离，不使用 SCMII 模式时，配置链接丢失响应时间可到 1.3 μs

表 4.2 ET1100 不兼容的 PHY 器件列表

制造商	器件	原因
AMD	Am79C874, Am79C875	根据数据手册或制造商提供数据，不支持 MDI/MDIX 自动交叉功能
Broadcom	BCM5208R	
Cortina Systems (Intel)	LXT970A, LXT971A, LXT972A, LXT972M, LXT974, LXT975	
Davicom 半导体	DM9761	
SMSC	LAN83C185	
ST 微电子	STE100P	
Teridian(TDK)	78Q2120C	
VIA 科技	VT61303F, VT6303L	
Marvell	88E3015, 88E3018	TX_CLK 相位不确定
Micrel	KS8041	硬件测试结果，没有前导位保持

4.2 微处理器操作的 EtherCAT 从站硬件设计实例

本书编者开发了一种使用 AVR 系列单片机控制的 EtherCAT 从站接口卡，使用 AVR 系列单片机 Atmega128 作为微处理器操作 ET1100，实现 EtherCAT 基本通信。其接口卡组成如图 4.1 所示，主要由以下 5 部分组成：

①EtherCAT 从站控制芯片 ET1100；

②Atmel 公司的 AVR 系列单片机 Atmega128，具有 4 K 字节内部 RAM 和 128 K 片上在线可编程 FLASH 存储器；

③以太网 PHY 芯片 KS8721；

④PULSE 公司以太网数据变压器 H1101；

⑤ RJ45 连接器。

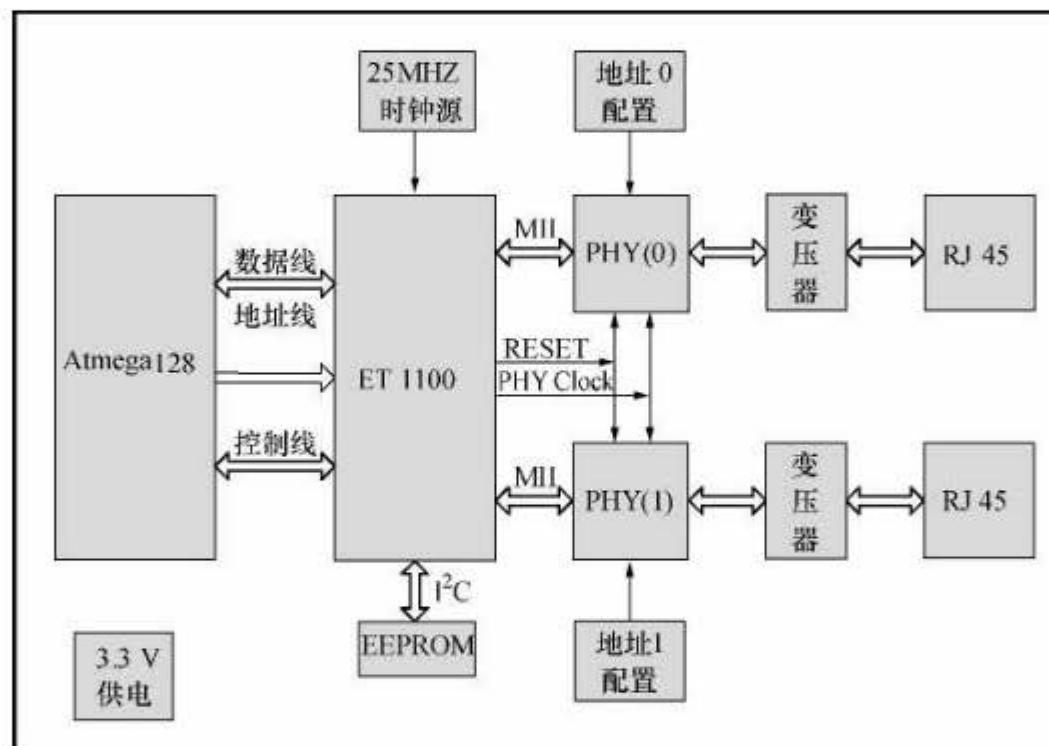


图 4.1 Atmega128 控制的 EtherCAT 从站接口卡示意图

4.2.1 ET1100 的接线

在本设计中，ET1100 使用 8 位异步微处理器 PDI 接口，连接两个 MII 接口，并输出时钟信号给 PHY 器件。图 4.2 给出了 ET1100、Atmega128 及 PHY 器件的连接图。图 4.3 给出了外部时钟源连接和 EEPROM 连接图。图 4.4 是 ET1100 的电源引脚连接，图 4.5 是 5 V~3.3 V 的变压电路。

此外，AVR 微处理器还连接着外部控制设备，例如 A/D 变换、D/A 变换和控制电机的脉冲信号输出。因本例的目的是重点介绍 AVR 微处理器对 ET1100 的操作，所以在图中略去了 AVR 与控制设备器件的连接。

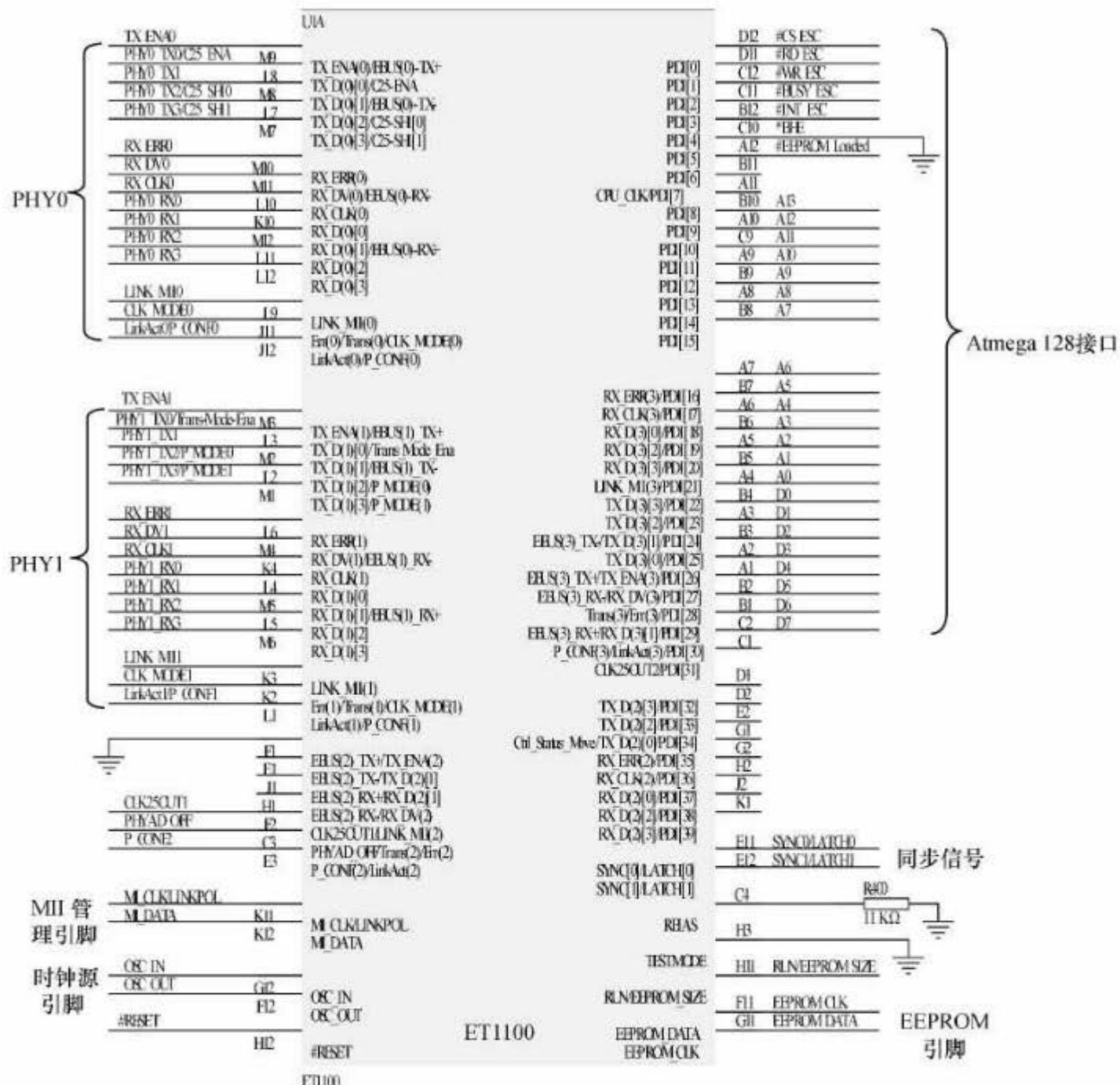


图 4.2 ET1100 连接图

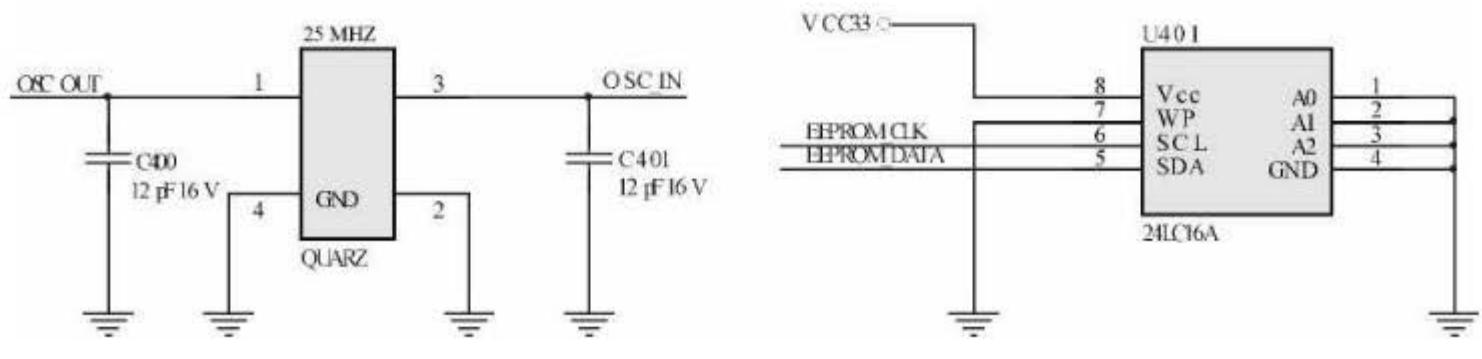


图 4.3 ET1100 时钟源和 EEPROM 接线图

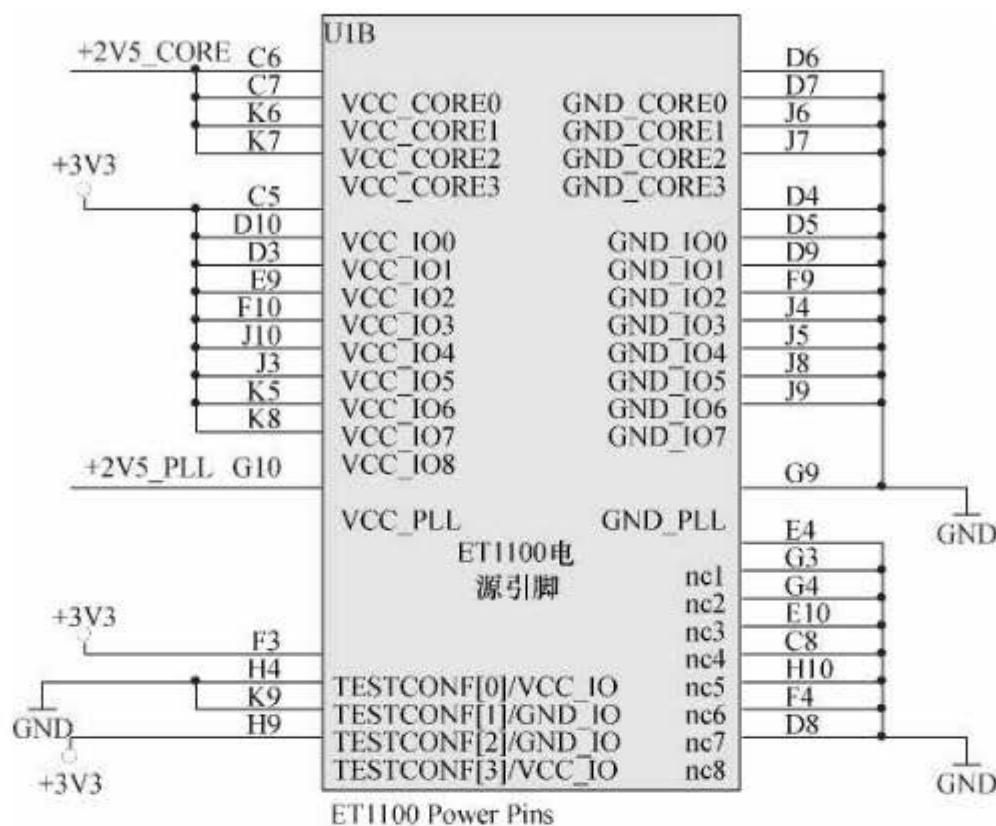


图 4.4 ET1100 电源引脚接线图

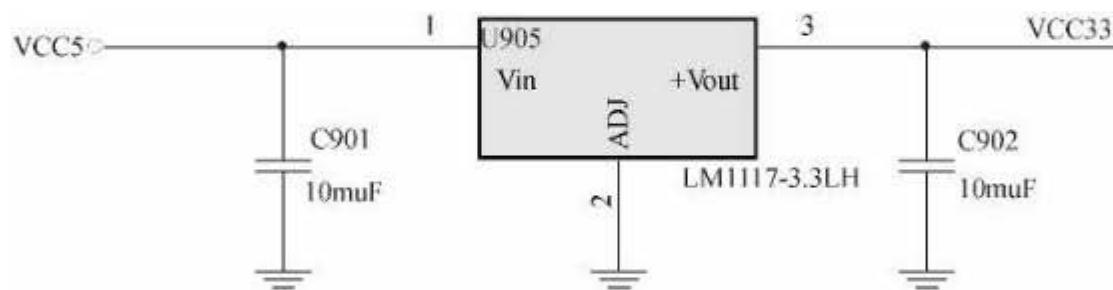


图 4.5 电压转换电路

4.2.2 ET1100 配置电路

ET1100 的配置引脚与 MII 引脚或其他引脚复用，在上电时作为输入由 ET1100 锁存配置信息。上电之后，这些引脚都有分配的操作功能，必要时引脚方向也可以改变。RESET 引脚信号指示上电配置完成。所有的配置引脚的取值和功能含义如表 4.3 所示。配置引脚的接线如图 4.6 所示，其中有些引脚具有 LED 状态输出功能。

表 4.3 ET1100 配置引脚配置值

编 号	名 称	引 脚	属性	取 值	含 义
1	C25_ENA	L8	I	0	不使能 CLK25OUT2 输出
2	C25_SHI[0]	L7	I	0	无 MII TX 相位偏移

续表 4.3

编 号	名 称	引脚	属性	取 值	含 义
3	C25_SHI[1]	M7	I	0	
4	CLK_MODE[0]	J11	I	0	
5	CLK_MODE[1]	K2	I	0	不输出 CPU 时钟信号
6	P_CONF(0)	J12	I	1	端口 0 使用 MII 接口
7	Trans_Mode_Ena	L3	I	0	不使用透明模式
8	P_MODE[0]	L2	I	0	
10	P_MODE[1]	M1	I	0	使用 ET1100 端口 0 和 1
11	P_CONF(1)	L1	I	0	端口 1 使用 MII 接口
12	PHYAD_OFF	C3	I	0	PHY 地址无偏移
13	LINKPOL	K11	I	0	LINK_MII(x)低有效

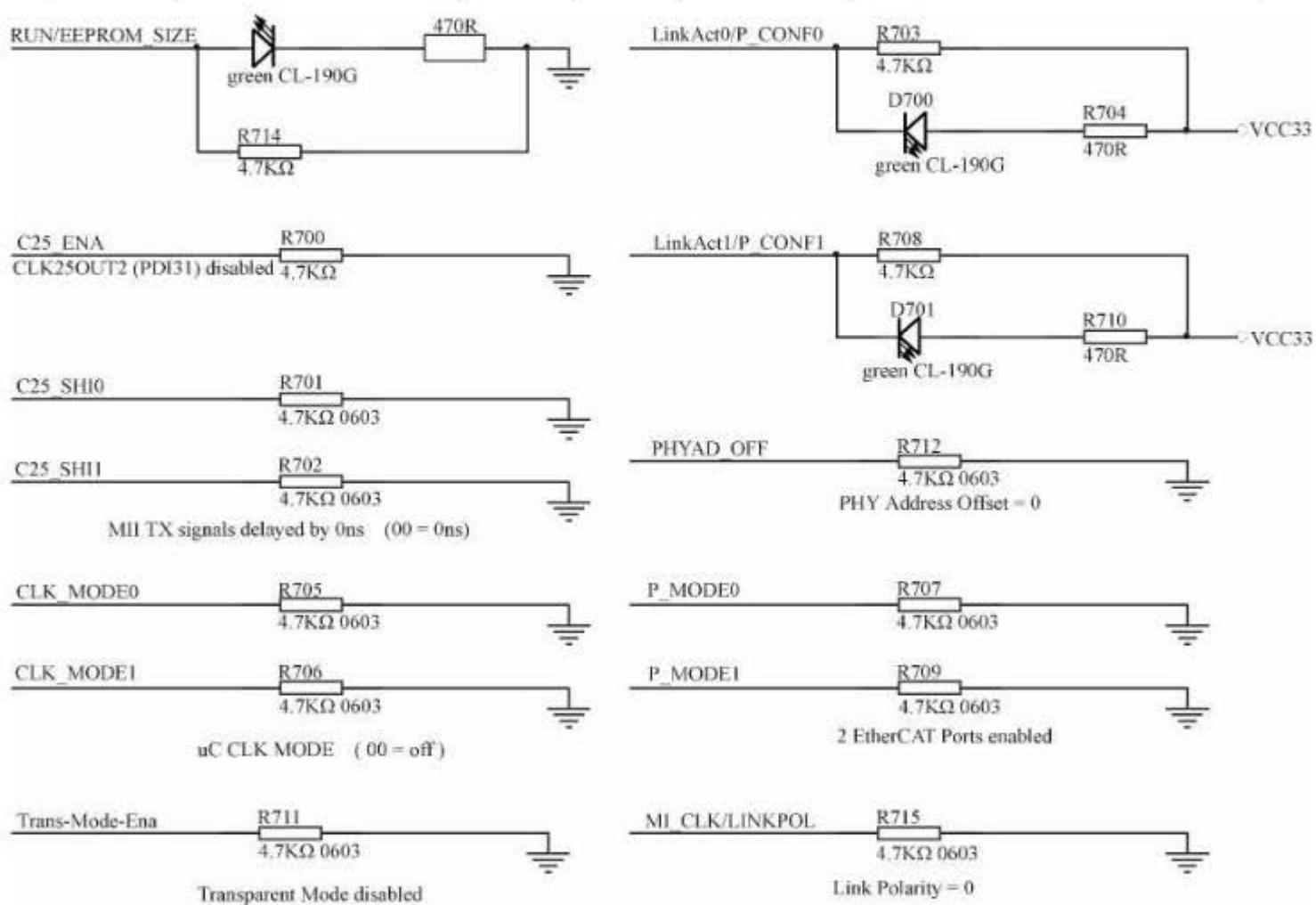


图 4.6 ET1100 配置引脚连接图

4.2.3 MII 接线

图 4.2 所示的 ET1100 接线中左半部分为 MII 接口相关引脚，包括两个 MII 端口引脚、相关 MII 管理引脚和时钟输出引脚等。表 4.4 列出了 MII 相关接线的详细说明。

表 4.4 MII 接线引脚说明

分类	编号	名称	引脚	属性	功能
端口 0	1	TX_ENA(0)	M9	O	端口 0 MII 发送使能
	2	TX_D(0)[0]	L8	O	端口 0 MII 发送数据 0
	3	TX_D(0)[1]	M8	O	端口 0 MII 发送数据 1
	4	TX_D(0)[2]	L7	O	端口 0 MII 发送数据 2
	5	TX_D(0)[3]	M7	O	端口 0 MII 发送数据 3
	6	RX_ERR(0)	M10	I	MII 接收数据错误指示
	7	RX_DV(0)	M11	I	MII 接收数据有效指示
	8	RX_CLK(0)	L10	I	MII 接收时钟
	9	RX_D(0)[0]	K10	I	端口 0 MII 接收数据 0
	10	RX_D(0)[1]	M12	I	端口 0 MII 接收数据 1
	11	RX_D(0)[2]	L11	I	端口 0 MII 接收数据 2
	12	RX_D(0)[3]	L12	I	端口 0 MII 接收数据 3
	13	LINK_MII(0)	L9	I	PHY0 指示有效连接
	14	LinkAct(0)	J12	O	LED 输出，链接状态显示
端口 1	1	TX_ENA(1)	M3	O	端口 1 MII 发送使能
	2	TX_D(1)[0]	L3	O	端口 1 MII 发送数据 0
	3	TX_D(1)[1]	M2	O	端口 1 MII 发送数据 1
	4	TX_D(1)[2]	L2	O	端口 1 MII 发送数据 2
	5	TX_D(1)[3]	M1	O	端口 1 MII 发送数据 3
	6	RX_ERR(1)	L6	I	MII 接收数据错误指示
	7	RX_DV(1)	M4	I	MII 接收数据有效指示
	8	RX_CLK(1)	K4	I	MII 接收时钟
	9	RX_D(1)[0]	L4	I	端口 1 MII 接收数据 0
	10	RX_D(1)[1]	M5	I	端口 1 MII 接收数据 1
	11	RX_D(1)[2]	L5	I	端口 1 MII 接收数据 2
	12	RX_D(1)[3]	M6	I	端口 1 MII 接收数据 3
	13	LINK_MII(1)	K3	I	PHY1 指示有效连接
	14	LinkAct(1)	L1	O	LED 输出，链接状态显示
其它	1	CLK25OUT1	F2	O	输出时钟信号给 PHY 芯片
	2	MI_CLK	K11		MII 管理接口时钟
	3	MI_DATA	K12		MII 管理接口数据

图 4.7 给出 ET1100 与 Micrel 公司的 PHY 器件 KS8721BL 连接的实例电路。KS8721BL 具有以下特点：

- ① 适用于 100BASE-TX/100BASE-FX/10BASE-T 物理层连接;
 - ② 2.5 V CMOS 设计, 2.5 V/3.3 V 的 I/O 电压;
 - ③ 3.3 V 单电源供电, 内部集成调压电路, 功率消耗小于 340mW (包括输出驱动电流);
 - ④ 完全符合 IEEE802.3 协议标准;
 - ⑤ 支持 MII 和 RMII 接口;
 - ⑥ 支持掉电和节电模式;
 - ⑦ 支持 10/100 Mbps 波特率和全/半双工模式的自动协商或设置选择;
 - ⑧ 片上集成前端模拟滤波器;
 - ⑨ 具有链接、活动、全/半双工、冲突和速度等指示 LED;
 - ⑩ 支持 MDI/MDIX 自动交叉;

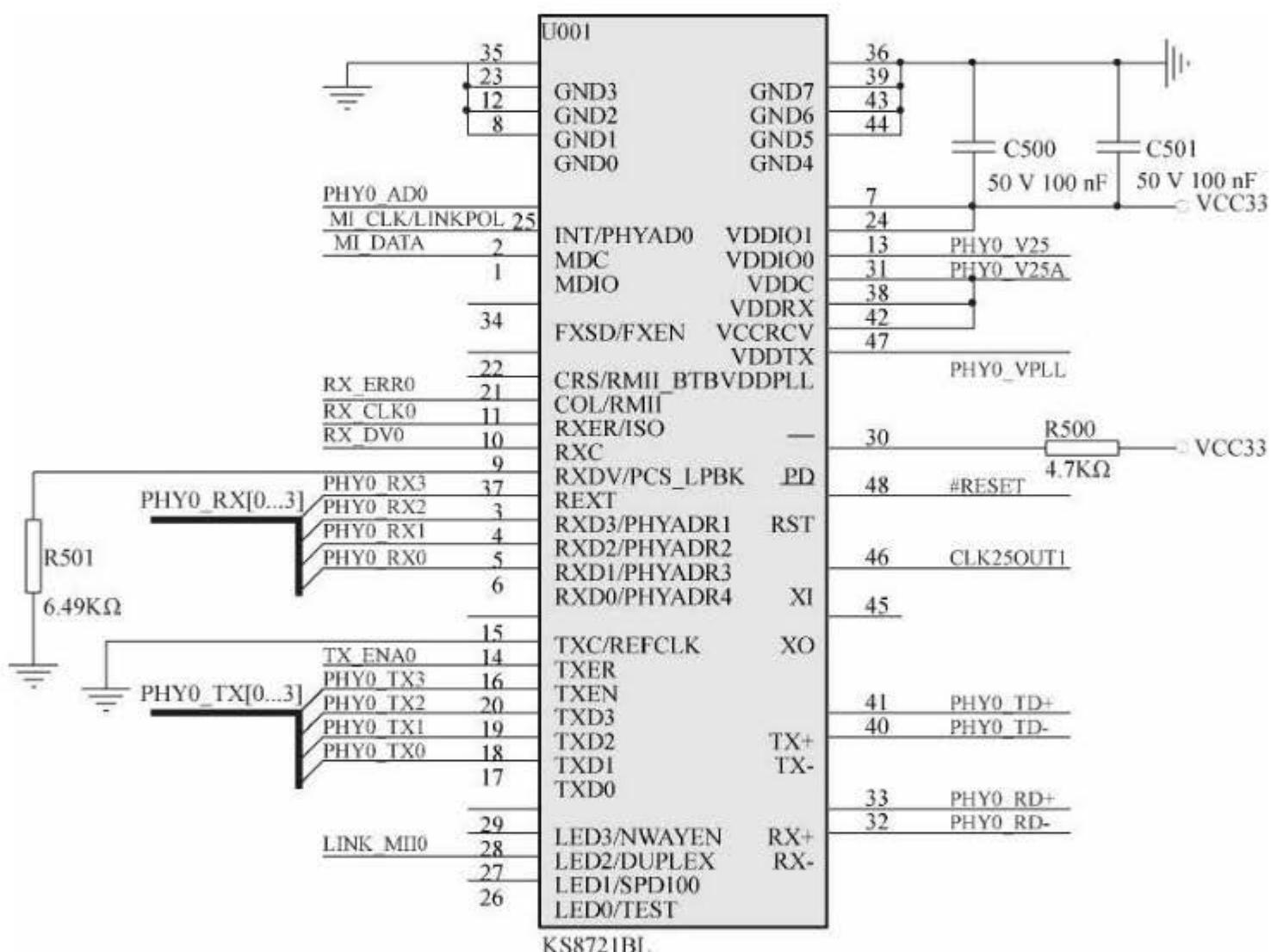


图 4.7 PHY 器件 KS8721 接线图

4.2.4 微处理器接口引脚接线

图 4.2 所示的 ET1100 接线中右半部分为微处理器接口引脚接线，包括 8 位数据线、14 位地址线和相关控制线。表 4.5 列出了这些接口及其接线的说明。

表 4.5 ET1100 与 8 位微处理器连接引脚

分 类	编 号	名 称	引 脚	属 性	功 能
数据 总 线	1	D0	B4	I/O	数据总线位 0
	2	D1	A3	I/O	数据总线位 1
	3	D2	B3	I/O	数据总线位 2
	4	D3	A2	I/O	数据总线位 3
	5	D4	A1	I/O	数据总线位 4
	6	D5	B2	I/O	数据总线位 5
	7	D6	B1	I/O	数据总线位 6
	8	D7	C2	I/O	数据总线位 7
地址 总 线	1	A0	A4	O	地址总线位 0
	2	A1	B5	O	地址总线位 1
	3	A2	A5	O	地址总线位 2
	4	A3	B6	O	地址总线位 3
	5	A4	A6	O	地址总线位 4
	6	A5	B7	O	地址总线位 5
	7	A6	A7	O	地址总线位 6
	8	A7	B8	O	地址总线位 7
	9	A8	A8	O	地址总线位 8
	10	A9	B9	O	地址总线位 9
	11	A10	A9	O	地址总线位 10
	12	A11	C9	O	地址总线位 11
	13	A12	A10	O	地址总线位 12
	14	A13	B10	O	地址总线位 13
控 制 线 和 状 态 线	1	*CS_ESC	D12	I	ET1100 片选
	2	*RD_ESC	D11	I	ET1100 读
	3	*WR_ESC	C12	I	ET1100 写
	4	*BUSY_ESC	C11	O	ET1100 操作忙
	5	*INT_ESC	B12	O	ET1100 中断，连接到 Atmega128 的外部中断输入引脚
	6	*BHE	C10	I	ET1100 高字节有效输入控制
	7	*EEPROM_Loaded	A12	O	ET1100 初始化完成信号
	8	SYNC0	E11	O	ET1100 同步信号 0
	9	SYNC1	E12	O	ET1100 同步信号 1

4.3 直接 IO 控制 EtherCAT 从站硬件设计实例

配置 ET1100 的 PDI 接口为 IO 控制, ET1100 可以直接控制 32 位数字量 I/O 信号。作者设计了一种 16 位数字量输入和 16 位数字量输出的 I/O 控制卡。其 MII 接口与 4.2.3 节介绍相同, PDI 接口直接当做 IO 信号使用, 如图 4.8 所示。由于 ET1100 使用 3.3 V 供电, 而外围电路为 5V 供电, 所以在 IO 引脚都串联了阻值 330 Ω 的电阻, 使电压匹配。

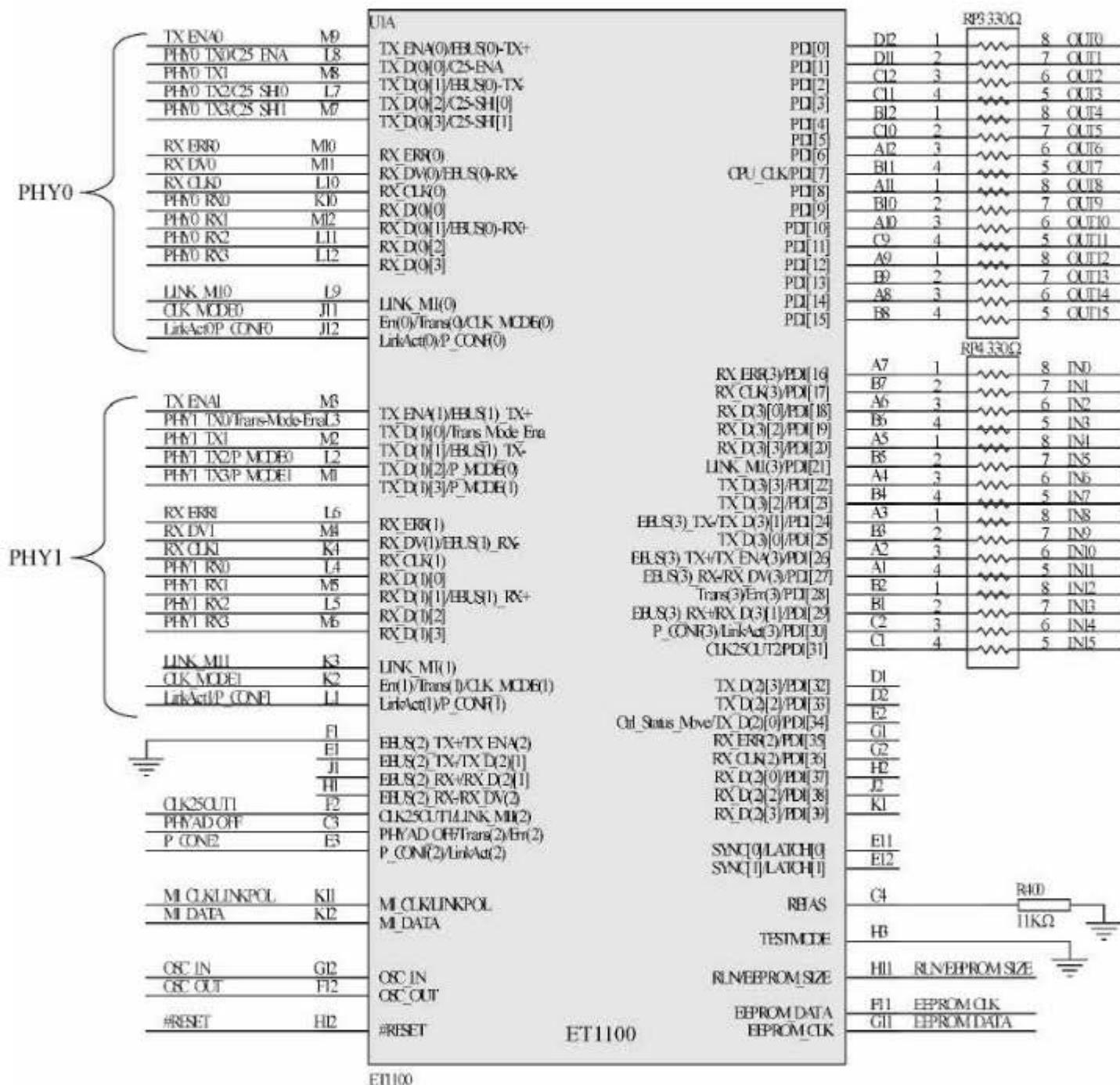


图 4.8 直接 IO 控制 ET1100 接线图

ET1100 的输入和输出引脚经过光电隔离后可以直接用于外部设备的控制。图 4.9 和图 4.10 分别是 8 位输出信号和 8 位输入信号的光电隔离接线图, 使用 TLP521-4 光电耦合器件。

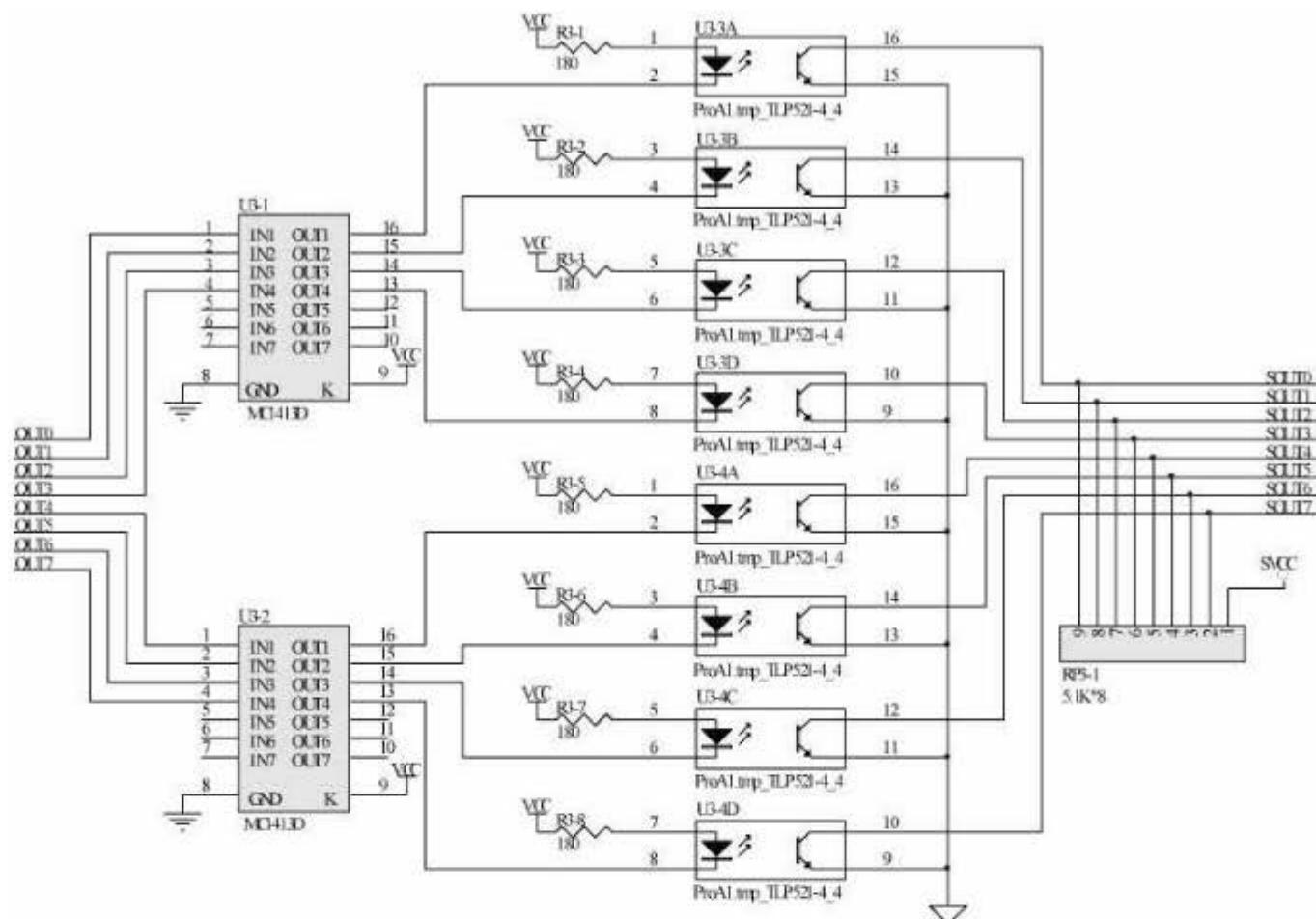


图 4.9 光电隔离输出信号接线

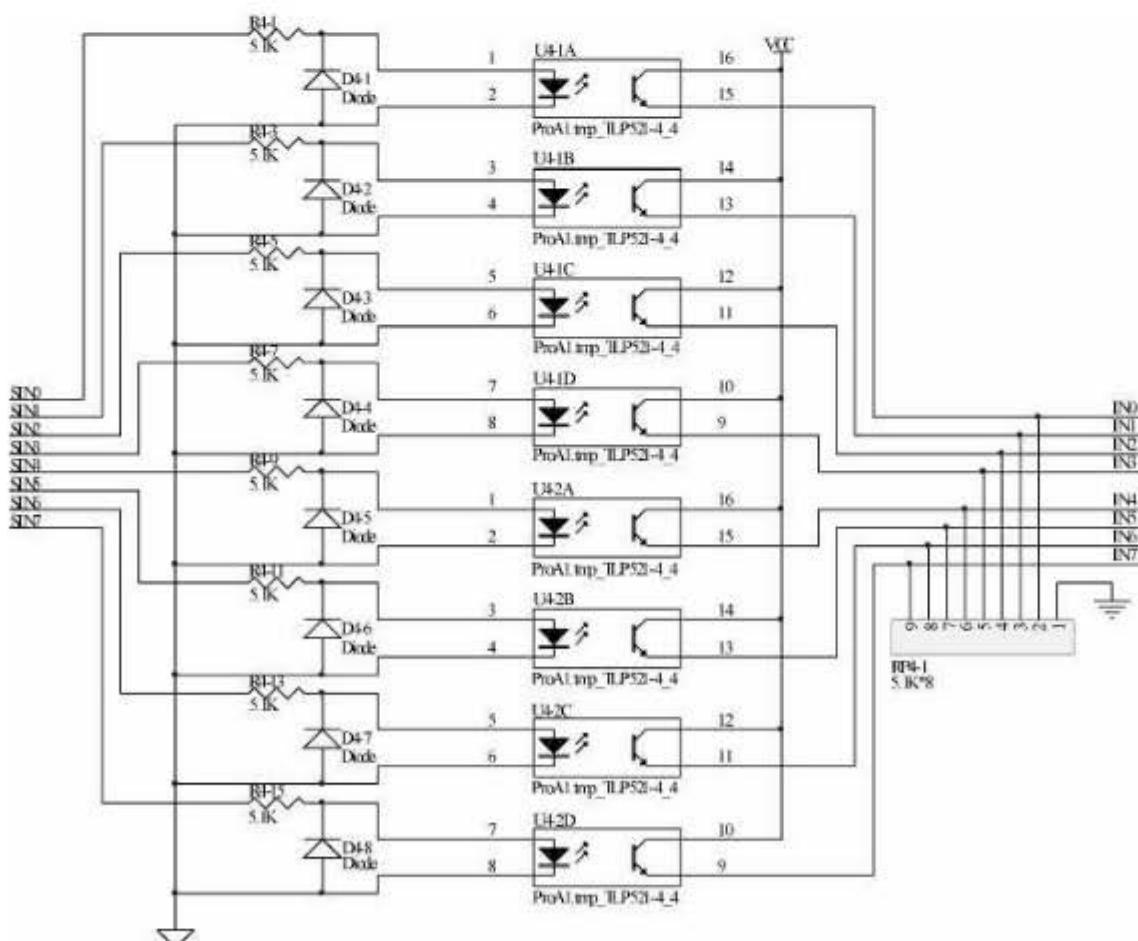


图 4.10 光电隔离输入信号接线

第5章 EtherCAT 伺服驱动器控制应用协议

IEC61800^[12-14]标准系列是一个可调速电子功率驱动系统通用规范。其中，IEC61800-7 定义了控制系统和功率驱动系统之间的通信接口标准，包括网络通信技术和应用行规，如图 5.1 所示。EtherCAT 作为网络通信技术，支持了 CANopen 协议中的行规 CiA 402 和 SERCOS 协议的应用层，分别称为 CoE 和 SoE。本章将分别介绍这两种 EtherCAT 应用层协议及其对应的伺服驱动器控制行规。

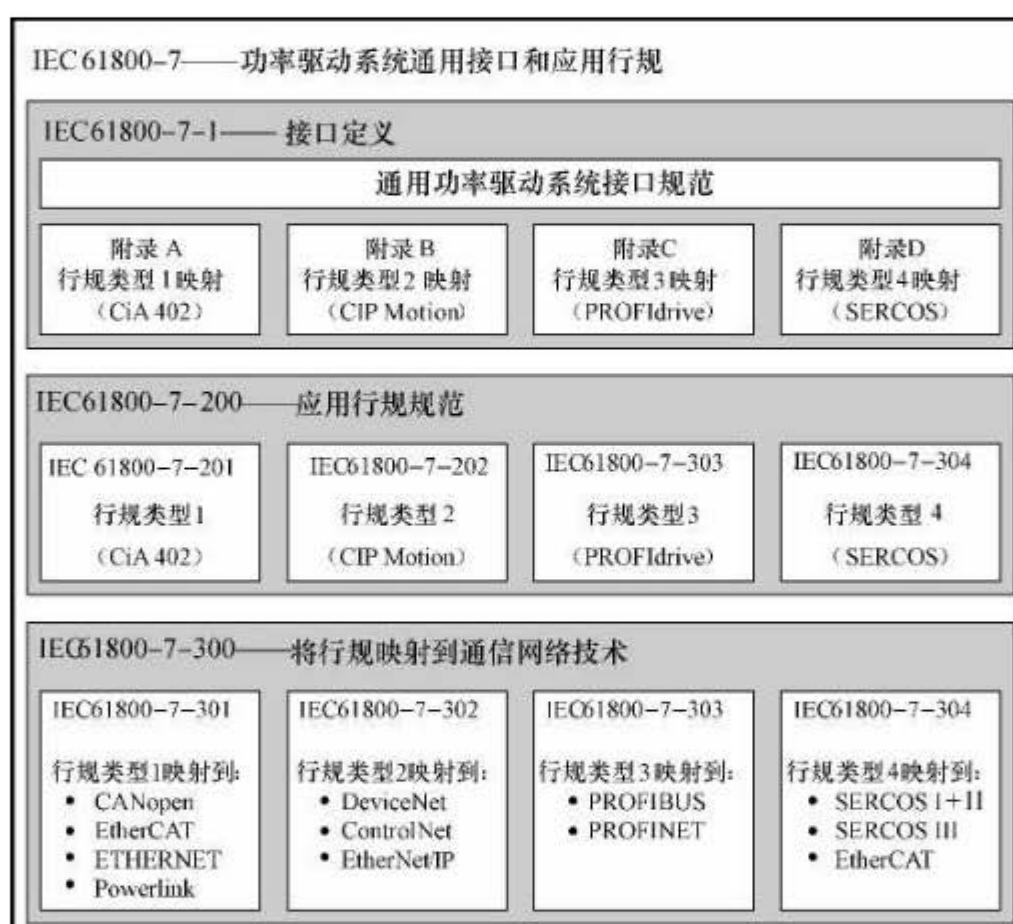


图 5.1 IEC61800-7 体系结构

5.1 CoE (CANopen over EtherCAT)

CANopen 最初是为基于 CAN (Control Area Network) 总线的系统所制定的应用层协议。EtherCAT 协议在应用层支持 CANopen 协议，并作了相应的扩充，其主要功能有：

- 使用邮箱通信访问 CANopen 对象字典及其对象，实现网络初始化；
- 使用 CANopen 应急对象和可选的事件驱动 PDO 消息，实现网络管理；
- 使用对象字典映射过程数据，周期性传输指令数据和状态数据。

5.1.1 CoE 对象字典

CoE 协议完全遵从 CANopen 协议，其对象字典的定义也相同，如表 5.1 所列。表 5.2 列出了 CoE 通信数据对象。其中针对 EtherCAT 通信扩展了相关通信对象 0x1C00~0x1C4F，用于设置存储同步管理器的类型、通信参数和 PDO 数据分配。

表 5.1 CoE 对象字典定义

索引号范围	含 义
0x0000~0xFFFF	数据类型描述
0x1000~0x1FFF	通信对象，它包括： • 设备类型、标识符、PDO 映射，与 CANopen 兼容 • CANopen 专用数据对象，在 EtherCAT 中保留 • EtherCAT 扩展数据对象
0x2000~0x5FFF	制造商定义对象
0x6000~0x9FFF	行规定义数据对象
0xA000~0xFFFF	保留

表 5.2 CoE 通信数据对象

索引号	含 义
0x1000	设备类型，32 位整数 位 0~15：所使用的设备行规 位 16~31：基于所使用行规的附加信息
0x1001	错误寄存器，8 位 位 0：常规错误 位 1：电流错误 位 2：电压错误 位 3：温度错误 位 4：通信错误 位 5：设备行规定义错误 位 6：保留 位 7：制造商定义错误
0x1008	设备商设备名称，字符串
0x1009	制造商硬件版本
0x100A	制造商软件版本
0x1018	设备标识符，结构体类型 子索引 0：参数体数目 子索引 1：制造商 ID (Vendor ID) 子索引 2：产品码 (Product Code) 子索引 3：版本号 (Revision Number) 子索引 4：序列号 (Serial Number)
0x1600~0x17FF	RxDPO 映射，结构体类型 子索引 0：参数体数目 子索引 1：第一个映射的输出数据对象 ： 子索引 n：最后一个映射的输出数据对象

续表 5.2

索引号	含 义
0x1A00~0x1BFF	TxDPO 映射, 结构体类型 子索引 0: 参数体数目 子索引 1: 第一个映射的输入数据对象 ⋮ 子索引 n: 最后一个映射的输入数据对象
0x1C00	同步管理器通信类型, 子索引 0 定义了所使用 SM 的数目, 子索引 1~32 定义了相应 SM0~SM31 通道的通信类型, 相关通信类型有, 0: 邮箱输出, 非周期性数据通信, 1 个缓存区写操作 1: 邮箱输入, 非周期性数据通信, 1 个缓存区读操作 2: 过程数据输出, 周期性数据通信, 3 个缓存区写操作 3: 过程数据输入, 周期性数据通信, 3 个缓存区读操作
0x1C10~0x1C2F	过程数据通信同步管理器 PDO 分配 子索引 0: 分配的 PDO 数目 子索引 1~n: PDO 映射对象索引号
0x1C30~0x1C4F	同步管理器参数 子索引 1: 同步类型 (见 2.5 节的 “1. 从站设备同步运行模式”) 子索引 2: 周期时间, 单位为 ns 子索引 3: AL 事件和相关操作之间的偏移时间, 单位为 ns

5.1.2 周期性过程数据通信

周期性数据通信中, 过程数据可以包含多个 PDO 映射数据对象, CoE 协议使用数据对象 0x1C10~0x1C2F 定义相应 SM 通道的 PDO 映射对象列表。以周期性输出数据为例, 输出数据使用 SM2 通道, 由对象数据 0x1C12 定义 PDO 分配, 如图 5.2 所示。表 5.3 列出了其取值实例。

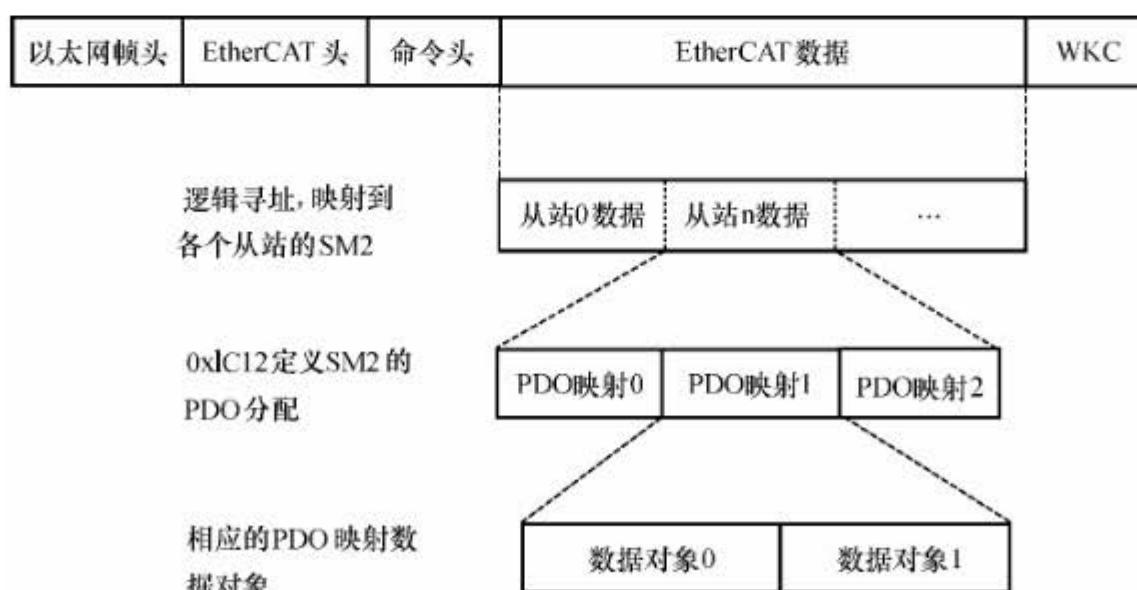


图 5.2 PDO 分配示意图

表 5.3 SM2 通道 PDO 分配对象数据 0x1C12 举例

子索引	数 值	PDO 数据对象映射			
		子索引	数 值	数据字节数	含 义
0	3			1	PDO 映射对象数目
1	PDO0 0x1600	0	2	1	数据映射数据对象数目
		1	0x7000:01	2	电流模拟量输出数据
		2	0x7010:01	2	电流模拟量输出数据
2	PDO1 0x1601	0	2	1	数据映射数据对象数目
		1	0x7020:01	2	电流模拟量输出数据
		2	0x7030:01	2	电流模拟量输出数据
3	PDO2 0x1602	0	2	1	数据映射数据对象数目
		1	0x7040:01	2	电流模拟量输出数据
		2	0x7050:01	2	电流模拟量输出数据

根据设备的复杂程度， PDO 过程数据映射又有以下几种形式：

- 使用固定的过程数据;
 - 在从站EEPROM中读取，不需要SDO协议。

(2) 可读取的PDO映射

 - 固定过程数据映射;
 - 可以使用SDO通信读取。

(3) 可选择的PDO映射

 - 多组固定的PDO通过PDO分配对象0x1C1x选择;
 - 通过SDO通信选择。

(4) 可变的PDO映射

 - 可通过CoE通信配置;
 - PDO内容可改变。

5.1.3 CoE 非周期性数据通信

EtherCAT 主站通过读写邮箱数据 SM 通道实现非周期性数据通信，邮箱数据定义如图 2.26，邮箱通信机制参见“2.5.2 非周期性邮箱数据通信”。CoE 协议邮箱数据结构如图 5.3 所示，其各元素定义如表 5.4 所列。

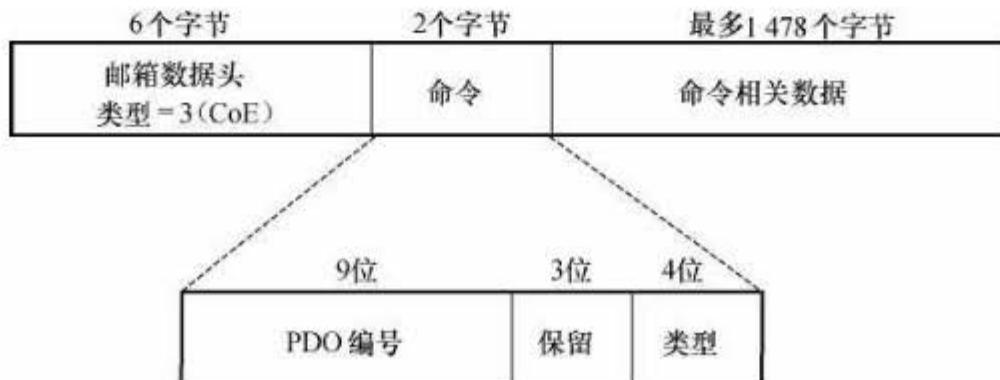


图 5.3 CoE 数据头

表 5.4 CoE 命令定义

数据元素	描述
PDO 编号	PDO 发送时的 PDO 序号
类 型	CoE 服务类型, 0: 保留 1: 紧急事件信息 2: SDO 请求 3: SDO 响应 4: TxPDO 5: RxPDO 6: 远程 TxPDO 发送请求 7: 远程 RxPDO 发送请求 8: SDO 信息 9~15: 保留

1. SDO 服务

CoE 通信服务类型 2 和 3 为 SDO 通信服务，SDO 数据结构如图 5.4 所示。SDO 通信服务的 3 种类型如图 5.5 所示。

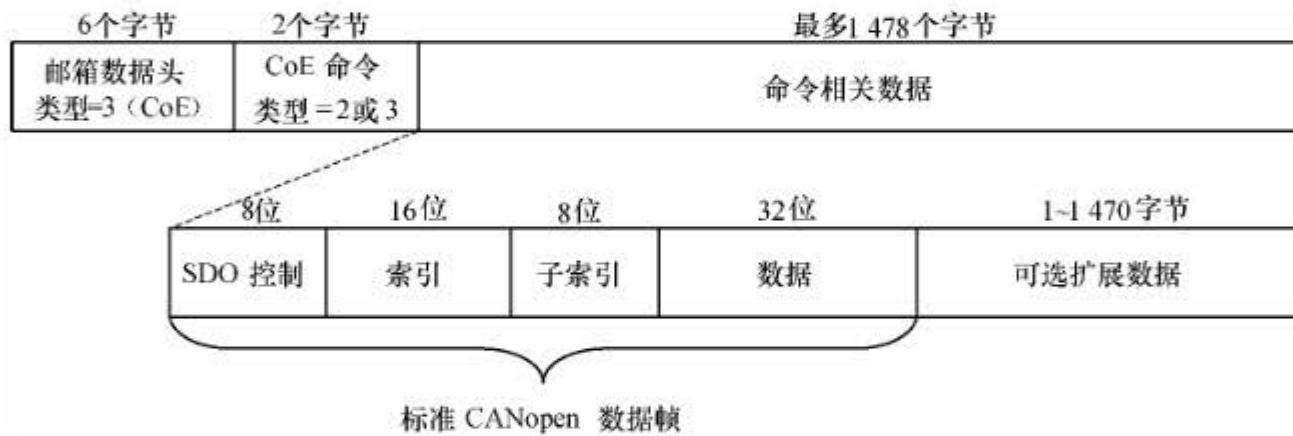


图 5.4 SDO 数据帧格式

①快速传输服务：与标准的 CANopen 协议相同，只使用 8 个字节，最多传输 4 个字节有效数据；

②常规传输服务：使用超过 8 个字节，可以传输超过 4 个字节的有效数据，最大可传输有效数据取决于邮箱 SM 所管理的存储区容量；

③分段传输服务：对于超过邮箱容量的情况，使用分段的方式进行传输。

SDO 数据帧格式如图 5.6 所示。

SDO 传输又分为下载和上传两种，下载传输常用于主站设置从站参数，上传传输用于主站读取从站的性能参数。这两种服务在物理上是对称的，本书只详细介绍下载传输，上传服务参见参考文献[5]和[6]中描述的 EtherCAT 应用层协议。

(1) SDO 下载传输请求

SDO 下载传输请求数据格式如图 5.6 所示。如果要传输的数据小于 4 个字节，则使用快速 SDO 传输服务，它完全兼容 CANopen 协议，使用 8 个字节数据，其中 4 个字节为数据区，有效字节数为 4 减去 SDO 控制字节中的位 2 和 3 表示的数值。

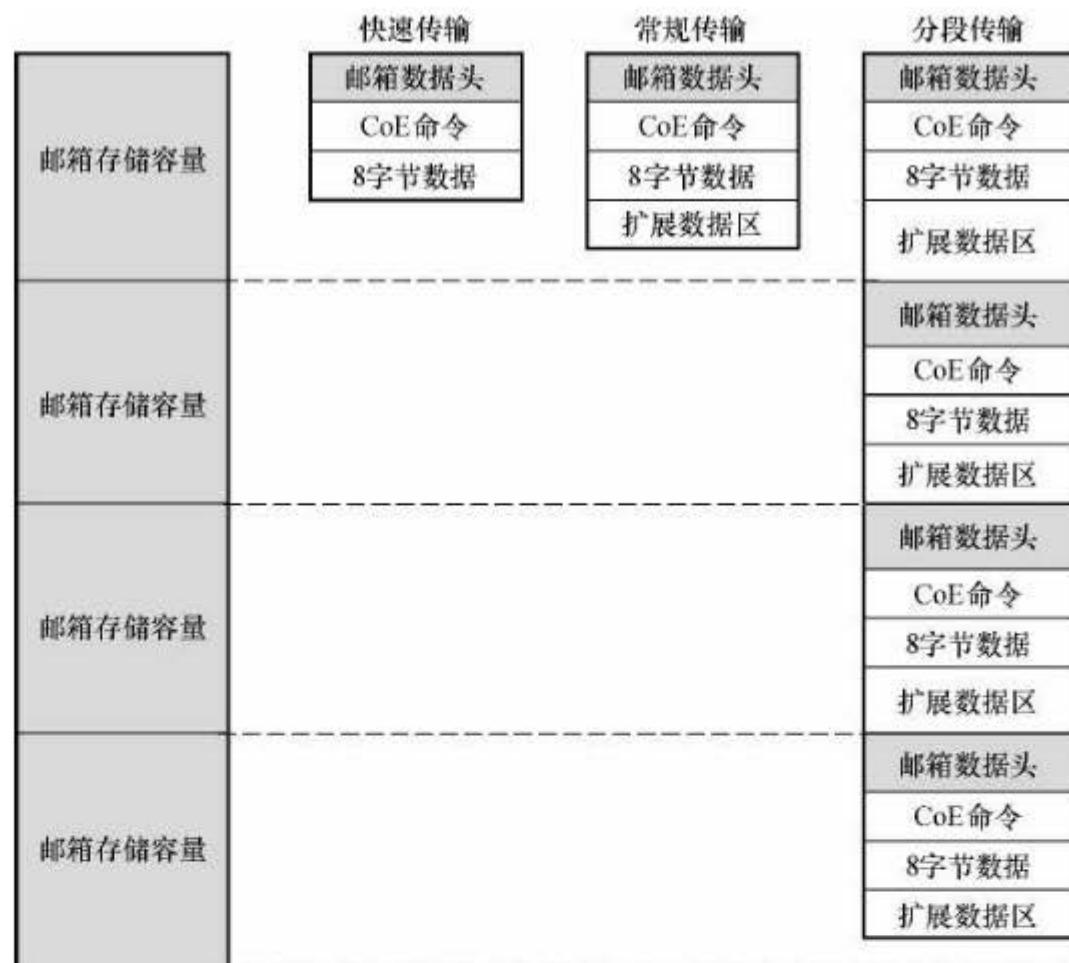


图 5.5 SDO 传输类型

如果要传输的数据大于 4 个字节，则使用常规传输服务。在常规传输时，用快速传输时的 4 个数据字节表示要传输的数据的完整大小，用扩展数据部分传输有效数据，有效数据的最大容量为邮箱容量减去 16，实际大小为邮箱头中长度数据 n 减去 10。SDO 下载传输请求服务的数据帧内容如表 5.5 所列。



图 5.6 SDO 下载服务数据帧格式

表 5.5 SDO 下载服务数据内容描述

数据区	字节数	位数	名称	取值和描述
邮箱头	2 字节	16 位	长度 n	$n \geq 0x0A$: 后续邮箱服务数据长度
	2 字节	16 位	地址	主站到从站通信, 为数据源从站地址 从站之间通信, 为数据目的从站地址
	1 字节	位 0~5	通道	0x00: 保留
		位 6~7	优先级	0x00: 最低优先级 ... 0x03: 最高优先级
	1 字节	位 0~3	类型	0x03: CoE
		位 4~7	保留	0x00
CoE 命令	2 字节	位 0~8	PDO 编号	0x00
		位 9~11	保留	0x00
		位 12~15	服务类型	0x02: SDO 请求
SDO 数据	1 字节 (控制字节)	位 0	数目指示 I Size Indicator	0x00: 未设置传输字节数目 0x01: 设置传输字节数目
		位 1	传输类型 E Transfer Type	0x01: 快速传输 0x00: 常规/分段传输
		位 2~3	传输字节数 Data Set Size	4-x: 快速传输时的有效数据字节数, x 是位 2~3 表示的数值 0: 常规/分段传输时无效
		位 4	完全操作 Complete Access	0x00: 操作由索引号和子索引号检索的参数体 0x01: 操作完整的数据对象, 子索引应该为 0 或 1 (不包括子索引 0)
		位 5~7	CoE 命令码 CCS (CoE Command Specifier)	0x01: 下载请求 0x00: 分段下载请求
	2 字节	16 位	索引号	数据对象索引号
	1 字节	8 位	子索引号	操作参数体子索引号
	4 字节	32 位	数据	快速传输: 数据 常规传输: 传输数据对象的总字节数, 如果本次传输的有效数据数目小于总数据长度, 则后续有分段传输数据
	$n-10$		扩展数据	常规传输的扩展数据, 传输有效数据

(2) SDO 分段下载传输

在常规下载传输时, 如果传输数据对象的总数量大于本次传输的允许数据数量, 则必须使用后续的分段下载传输服务, 数据帧格式如图 5.6 所示, 其数据元素描述如表 5.6 所列。

表 5.6 分段下载服务数据内容描述

数据区	字节数	位 数	名 称	取值和描述
邮箱头	2字节	16 位	长度	$n \geq 0x0A$: 后续邮箱服务数据长度
	2字节	16 位	地址	主站到从站通信, 为数据源从站地址 从站之间通信, 为数据目的从站地址
	1字节	位 0~5	通道	0x00: 保留
		位 6~7	优先级	同表 5.5
	1字节	位 0~3	类型	0x03: CoE
		位 4~7	保留	0x00
CoE 命令	2字节	位 0~8	PDO 编号	0x00
		位 9~11	保留	0x00
		位 12~15	服务类型	0x02: SDO 请求
SDO 控制 数据	1字节	位 0	是否有后续分段	0x00: 有后续传输分段 0x01: 最后一个下载分段
		位 1~3	分段数据数目 SegData Size	7-x: 最后 7 个字节中的有效数据数目, x 是位 1~3 表示的数值
		位 4	翻转握手位 Toggle	每次在 SDO 下载分段请求时翻转, 从 0x00 开始
		位 5~7	CoE 命令码 CCS (CoE Command Specifier)	0x01: 下载请求 0x00: 分段下载请求
	$n-3$		数据	分段传输数据

(3) SDO 下载传输响应

从站收到 SDO 下载请求之后, 执行相应处理, 然后将响应数据写入输入邮箱 SM1 中, 由主站读走。主站只有得到正确的响应之后才能执行下一步 SDO 操作。正确的 SDO 下载响应数据格式如图 5.7 所示, 其各元素描述如表 5.7 所列。

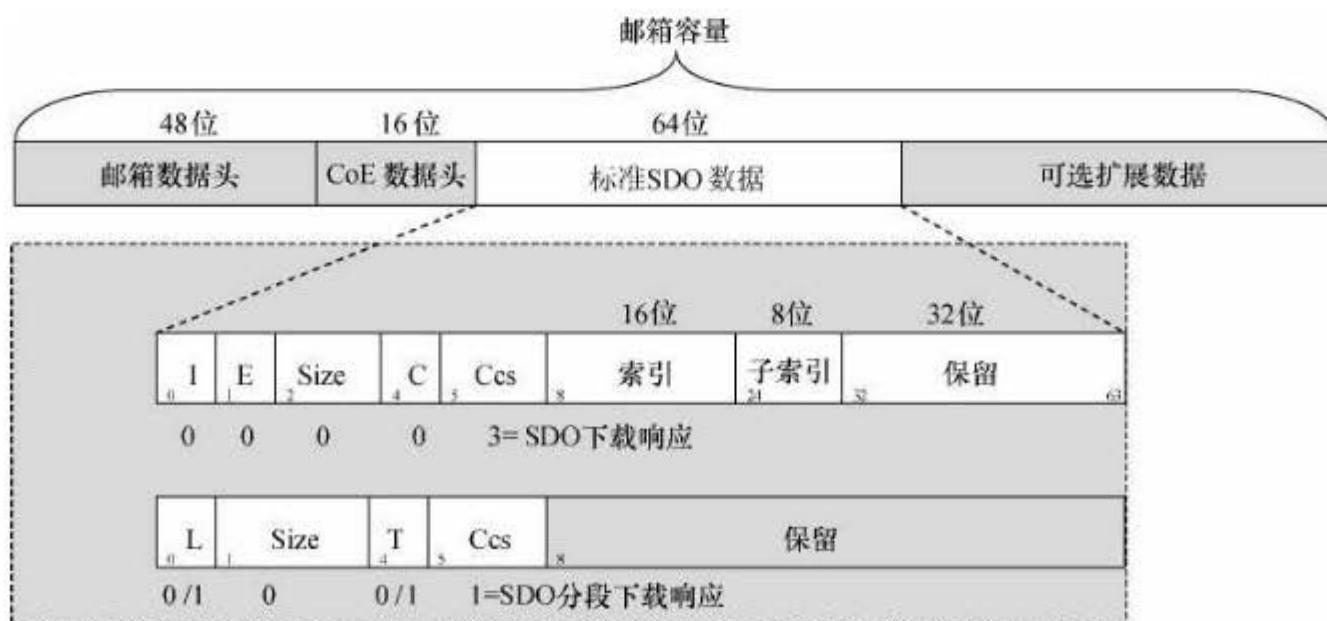


图 5.7 SDO 下载响应数据格式

表 5.7 SDO 下载响应数据描述

数据区	字节数	位数	名称	取值和描述
邮箱头	2 字节	16 位	长度	$n \geq 0x0A$: 后续邮箱服务数据长度
	2 字节	16 位	地址	主站到从站通信, 为数据源从站地址 从站之间通信, 为数据目的从站地址
	1 字节	位 0~5	通道	0x00: 保留
		位 6~7	优先级	同表 5.5
	1 字节	位 0~3	类型	0x03: CoE
		位 4~7	保留	0x00
CoE 命令	2 字节	位 0~8	PDO 编号	0x00
		位 9~11	保留	0x00
		位 12~15	服务类型	0x03: SDO 响应
快速和正常下载响应 SDO				
快速和正常传输 SDO 数据	1 字节	位 0	数目指示 I	0x00
		位 1	传输类型 E	0x00
		位 2~3	传输数目	0
		位 4	完全操作	同表 5.5
		位 5~7	CoE 命令码 CCS	0x03: 下载响应 0x01: 分段下载响应
	2 字节	16 位	索引号	数据对象索引号
	1 字节	8 位	子索引号	操作参数体子索引号
	4 字节	32 位	保留	保留
分段下载响应 SDO				
分段下载响应 SDO	1 字节	位 0~3	保留	0x00
		位 4	翻转位	与相应的分段下载请求相同
		位 5~7	CoE 命令码 CCS	0x03: 下载响应
	7 字节		保留	

(4) 终止 SDO 传输

在 SDO 传输过程中, 如果某一方发现有错误, 可以发起 SDO 终止传输请求, 对方收到此请求后, 停止当前 SDO 传输。**SDO 终止传输请求不需要应答**。表 5.8 描述了该请求的相关数据元素。其中 SDO 数据有 4 个字节的终止代码, 表示了终止传输的具体原因, 如表 5.9 所列。

表 5.8 SDO 终止传输请求数据描述

数据区	字节数	位数	名称	取值和描述
邮箱头	2 字节	16 位	长度	$n=0x0A$: 后续邮箱服务数据长度
	2 字节	16 位	地址	主站到从站通信, 为数据源从站地址 从站之间通信, 为数据目的从站地址
	1 字节	位 0~5	通道	0x00: 保留
		位 6~7	优先级	同表 5.5
	1 字节	位 0~3	类型	0x03: CoE
		位 4~7	保留	0x00
CoE 命令	2 字节	位 0~8	PDO 编号	0x00
		位 9~11	保留	0x00
		位 12~15	服务类型	0x02: SDO 请求
SDO 数据	1 字节 (控制字节)	位 0	数目指示 I	0x00
		位 1	传输类型 E	0x00: 常规/分段传输
		位 2~3	传输数目	0x00: 常规/分段传输, 无效
		位 4	保留	
		位 5~7	CoE 命令码 CCS	0x04: 终止传输请求
	2 字节	16 位	索引号	数据对象索引号
	1 字节	8 位	子索引号	操作参数体子索引号
	4 字节	32 位	终止代码	表示终止传输的原因, 见表 5.9

表 5.9 终止 SDO 传输代码

序号	代码值	含义
1	0x05 03 00 00	分段传输时翻转位无变化
2	0x05 04 00 00	SDO 传输超时
3	0x05 04 00 01	命令码无效或未知
4	0x05 04 00 05	内存溢出
5	0x06 01 00 00	不支持对某一对象的操作
6	0x06 01 00 01	读一个只写数据对象
7	0x06 03 00 02	写一个只读数据对象
8	0x06 02 00 00	数据对象在数据字典中不存在
9	0x06 04 00 41	数据对象不能映射到 PDO 中
10	0x06 04 00 42	要映射的数据对象的数量和长度超过了 PDO 数据长度
11	0x06 04 00 43	常规的参数不兼容
12	0x06 04 00 47	设备中常规内部不兼容
13	0x06 06 00 00	由于硬件错误导致操作失败

续表 5.9

序号	代码值	含 义
14	0x06 07 00 10	数据类型不匹配，服务参数长度不匹配
15	0x06 07 00 12	数据类型不匹配，服务参数长度过长
16	0x06 07 00 13	数据类型不匹配，服务参数长度过短
17	0x06 09 00 11	子索引不存在
18	0x06 09 00 30	写操作时，写入数据值超出范围
19	0x06 09 00 31	写入数据值太大
20	0x06 09 00 32	写入数据值太小
21	0x06 09 00 36	最大值小于最小值
22	0x08 00 00 00	普通错误
23	0x08 00 00 20	数据不可以被传输或保存到应用程序
24	0x08 00 00 21	由于本地控制原因，数据不可以被传输或保存到应用程序
25	0x08 00 00 22	由于当前设备状态原因，数据不可以被传输或保存到应用程序
26	0x08 00 00 23	对象字典动态生成错误，或没有找到对象字典

(5) SDO 下载传输举例

图 5.8 为 SDO 快速下载传输实例，主站要下载的有效数据小于 4 个字节，使用快速传输服务。主站首先发送快速 SDO 下载请求到从站 SM0，从站读取邮箱数据后执行相应操作，并将响应数据写入输入邮箱 SM1。主站读 SM1，读到有效数据后，根据响应数据判断下载请求的执行结果。图 5.8 中箭头表示有效数据的方向，从站到主站的有效数据也需要由主站发送读数据子报文来读取。

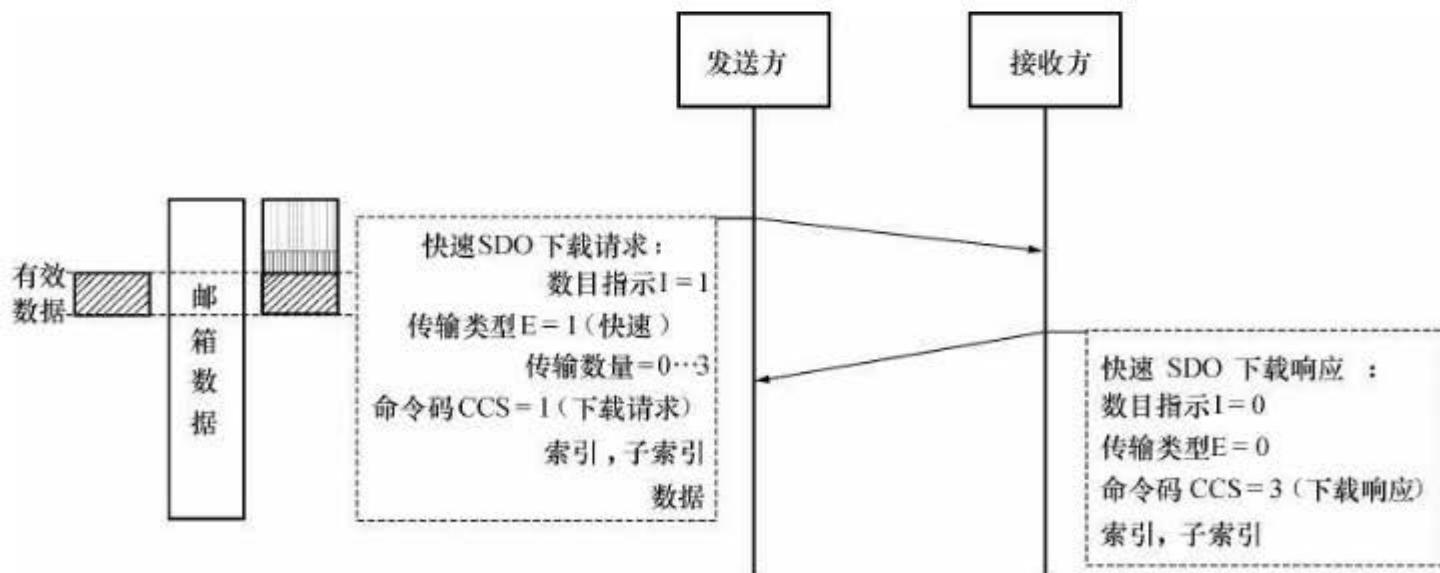


图 5.8 SDO 快速下载传输实例

图 5.9 为 SDO 常规下载传输实例，主站要下载的有效数据大于 4 个字节且小于邮箱容量，所以使用扩展数据区进行传输，其传输过程和快速下载传输类似。

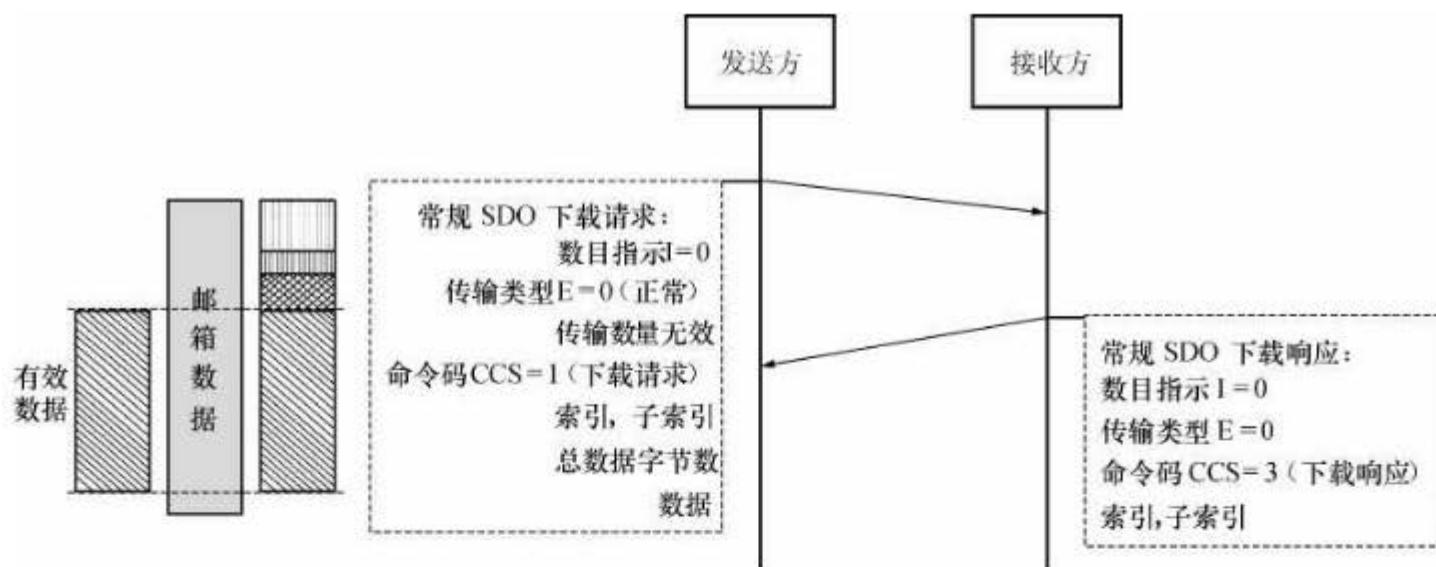


图 5.9 常规 SDO 下载传输实例

图 5.10 为 SDO 分段下载传输实例，主站要下载的有效数据大于邮箱容量，所以必需分段传输，每一个传输步骤都必需得到正确的响应才能继续后续步骤。

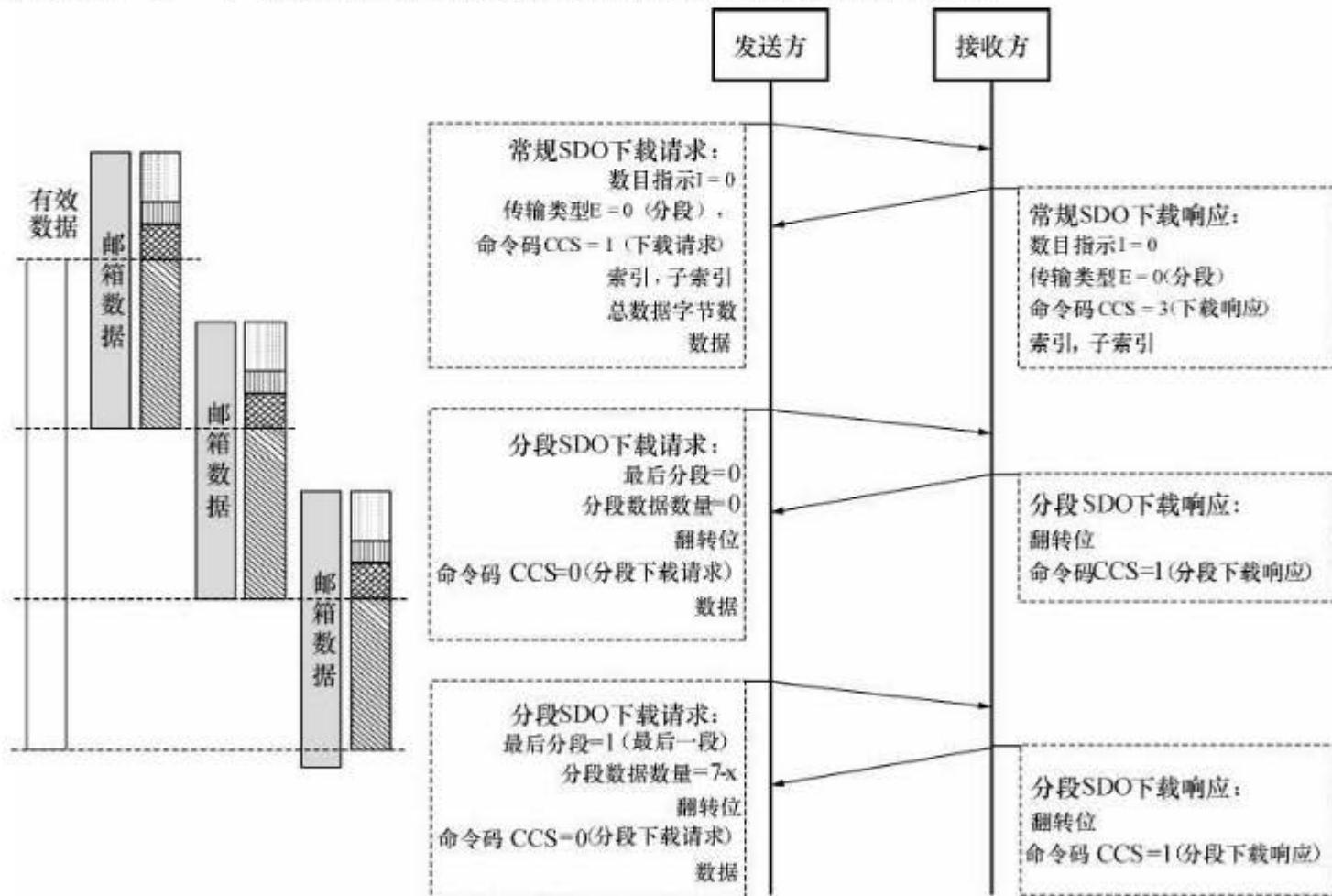


图 5.10 SDO 分段下载传输实例

在传输的过程中，如果主站或从站发现错误，则发起终止 SDO 传输请求，另一方收到此请求后即停止当前传输过程。

2. 紧急事件

紧急事件由设备内部的错误事件触发，将诊断信息发送给主站。当诊断事件消失之后，

从站应该将诊断事件和错误复位码再发送一次。紧急事件数据帧格式如图 5.11 所示，其各个数据元素描述如表 5.10 所列。



图 5.11 紧急事件数据帧格式

表 5.10 紧急事件数据元素描述

数据区	字节数	位数	名称	取值和描述
邮箱头	2 字节	16 位	长度	$n = 0x0A$: 后续邮箱服务数据长度
	2 字节	16 位	地址	主站到从站通信, 为数据源从站地址 从站之间通信, 为数据目的从站地址
	1 字节	位 0~5	通道	0x00: 保留
		位 6~7	优先级	同表 5.5
	1 字节	位 0~3	类型	0x03: CoE
		位 4~7	保留	0x00
CoE 命令	2 字节	位 0~8	PDO 编号	0x00
		位 9~11	保留	0x00
		位 12~15	服务类型	0x01: 紧急数据
SDO 控制 数据	2 字节	16 位	紧急错误码	见表 5.11
	1 字节	8 位	错误寄存器	映射数据对象 0x1001
	5 字节	40 位	数据	制造商定义错误信息

紧急事件的错误码定义如表 5.11 所列。

表 5.11 错误码定义

错误码 (16 进制)	含义
00xx	错误复位或无错误
10xx	常规错误
20xx	电流错误： 21xx 设备输入端电流错误 22xx 设备内部电流错误 23xx 设备输出端电流错误

续表 5.11

错误码 (16 进制)	含 义
30xx	电压错误: 31xx 主电路电压错误 32xx 设备内部电压错误 33xx 输出电压错误
40xx	温度错误: 41xx 环境温度 42xx 设备温度
50xx	设备硬件错误
60xx	设备软件错误: 61xx 内部软件错误 62xx 用户软件错误 63xx 数据错误
70xx	附加模块错误
80xx	监控: 81xx 通信错误 82xx 协议错误: 8210PDO 由于长度错误而未被处理 8220PDO 超长
90xx	外部错误
A0xx	EtherCAT 状态机错误
F0xx	附加功能
FFxx	设备自定义

5.1.4 应用层行规

CoE 完全遵从 CANopen 的应用层行规, CANopen 标准应用行规主要有:

- ① CiA 401 I/O 模块行规;
- ② CiA 402 伺服和运动控制行规;
- ③ CiA 403 人机接口行规;
- ④ CiA 404 测量设备和闭环控制;
- ⑤ CiA 406 编码器;
- ⑥ CiA 408 比例液压阀等。

本节介绍 CiA 402——伺服和运动控制行规。

1. CiA 402 行规通用数据对象字典

数据对象 0x6000~0x9FFF 为 CANopen 行规定义数据对象, 一个从站最多控制 8 个伺服驱动器, 每个驱动器分配 0x800 个数据对象。第一个伺服驱动器使用 0x6000~0x7FF 的数据

字典范围，后续伺服驱动器在此基础上以 0x800 偏移使用数据字典。每个内部模块的数据对象号等于 $0x6xxx + n \times 0x800$ ，CiA 402 基本数据对象如表 5.12 所列。

表 5.12 CiA 402 基本数据对象

索引号	类型	含义及取值
0x6402	16 位整型	从站控制电机类型， 0: 非标准电机 1: 2: 3: 永磁同步电动机 4: 变频控制同步电动机 5: 开关磁阻电动机 6: 交流异步绕线转子电动机 7: 鼠笼式交流异步电动机 8: 步进电动机 9: 细分步进电动机 10: 正弦波永磁无刷电动机 11: 方波永磁无刷电动机 12: 交流同步磁阻电动机 13: 直流永磁电动机 14: 直流串励电动机 15: 直流并励电动机 16: 直流复励电动机
0x6403	字符串	由制造商提供的电动机规格代码 (catalogue number)
0x6404	字符串	电机制造商名称
0x6405	字符串	电机样本网址
0x6406	日期	电机上次检测的日期
0x6407	16 位整型	电机的服务周期
0x6503	字符串	驱动器规格代码 (catalogue number)
0x6505	字符串	驱动器制造商的网址

2. 功率驱动设备的控制状态机

CiA 402 定义功率驱动设备的控制状态机如图 5.12 所示。表 5.13 详细描述了每个状态改变步骤的触发事件和所执行操作。只有相关操作正确完成之后才能切换到新的状态。

- (1) 图 5.12 中虚线的状态为可选项，细线框的状态可以由从站主动切换，而粗线框内的状态必须由主站检查和切换；
- (2) 区域 A 中只有低级别电源使能，高级别电源也有可能为了给低级别电源供电而使能；低级别电源指 24 V 等控制部分的供电，高级别电源指 230 V 或 380 V 的主电路供电；
- (3) 区域 B 中高级别供电有效，但是电动机没有扭矩输出，此时设定的指令数据应该被忽略；
- (4) 区域 C 中的状态为：伺服设备准备好运行，电动机有扭矩输出，设定的指令数据应该被执行。

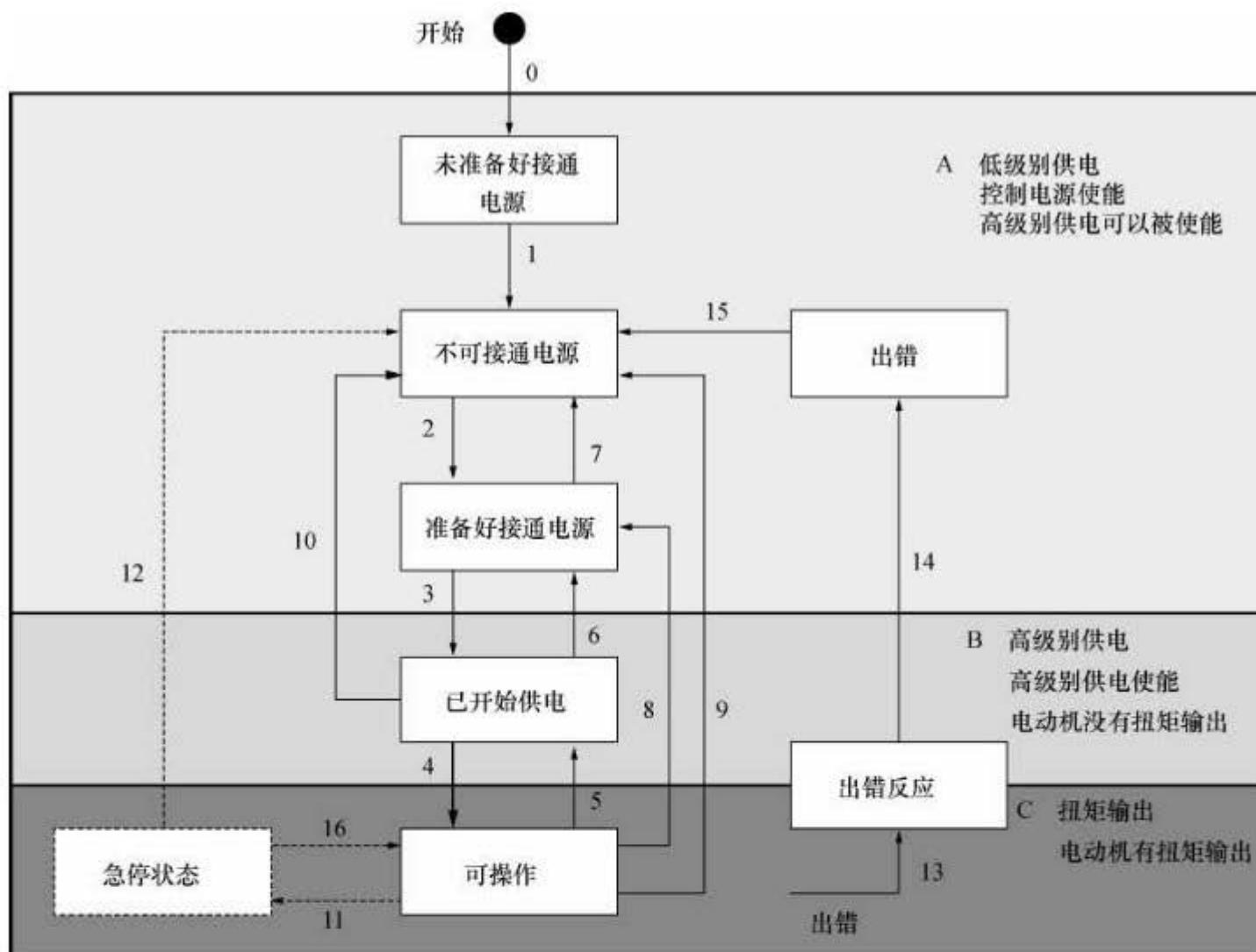


图 5.12 功率驱动设备控制有限状态机

表 5.13 状态转化触发事件和执行操作

状态转化	触发事件	执行操作
0	上电或复位后自动转化	伺服设备自检, 如果需要, 则执行相关初始化动作
1	自动转化	启动通信功能
2	从主站获得切断电源命令	无
3	从主站获得接通电源命令	如果条件满足, 则接通高级别电源
4	从主站获得使能运行命令	使能伺服功能, 清除内部设定指令
5	从主站获得停止运行命令	停止伺服运行功能
6	从主站获得切断电源命令	如果条件满足, 则切断高级别电源
7	从主站获得急停或停止供电命令	无
8	从主站获得切断电源命令	伺服功能失效, 如果条件满足, 则切断高级别电源
9	从主站获得停止供电命令	伺服功能失效, 如果条件满足, 则切断高级别电源
10	从主站获得急停或停止供电命令	如果条件满足, 则切断高级别电源
11	从主站获得急停命令	启动紧急停止功能

续表 5.13

状态转化	触发事件	执行操作
12	急停功能执行完成后且快速停止选项码为 1、2、3、4，或从主站收到关断电源命令	停止伺服驱动功能，如果条件满足，则切断高级别电源
13	出错	执行相应配置的错误反应功能
14	自动转换	停止伺服驱动功能，如果条件满足，则切断高级别电源
15	从主站接收错误复位命令	设备无错误存在时，执行错误条件复位；离开出错状态后，控制字中的错误复位位应该被控制设备清除
16	如果快速停止命令选项码为 5、6、7 或 8，则收到使能运行命令后执行	使能驱动功能（不推荐使用这个转化）

主站通过写控制字给从站来控制从站的状态，从站通过状态字来反馈自己的当前状态。控制字由数据对象 0x6040 定义，如表 5.14 所列。状态字由数据对象 0x6041 定义（如表 5.15 所列），其分类也沿用了 CANopen 的定义，M 表示必须的（Mandatory），C 表示有条件的（Conditional），O 表示可选的（Optional），R 表示推荐的（Recommended）。

表 5.14 控制字数据对象 0x6040 定义

位	含义	分 类	备 注
0	接通电源	M	0→1：接通电源，对应转化 3 1→0：切断电源，对应转化 2、6、8
1	使能供电	M	0→1：使能供电，对应转化 3 1→0：停止供电，对应转化 7、9、10、12
2	紧急停止	C	1→0：紧急停止，支持急停状态时有效，对应转化 7、10、11
3	使能运行	M	0→1：使能运行，对应转化 4、16 1→0：停止运行，对应转化 5
4-6	运行模式相关	O	
7	复位错误	M	对应转化 15
8	暂停	O	
9	运行模式相关	O	
10	保留	O	
11-15	制造商自定义	O	

表 5.15 状态字数据对象 0x6041 定义

位	含义	分 类	备 注
0	准备好接通电源	M	1：已准备好接通电源
1	电源已接通状态	M	1：电源已经接通
2	运行使能状态	M	1：运行已使能

续表 5.15

位	含义	分类	备注
3	出错状态	M	1: 已出错
4	电源使能状态	O	1: 高级别电源使能
5	急停状态	C	0: 处于急停状态 1: 不支持急停或急停功能没有运行
6	不可接通状态	M	0: 处于不可接通电源状态
7	报警	O	1: 发生报警
8	制造商定义	O	
9	远程	O	1: 控制字被处理 0: 控制字未被处理
10	目标指令到达	O	1: 达到目标指令值
11	内部限制启动	O	1: 超过内部极限而不能达到目标指令值, 例如, 硬件限位开关, 电流限制, 或热过载
12	放弃目标指令	M	1: 驱动器由于本地原因不能跟随目标值
13	运行状态定义	O	
14~15	制造商定义	O	

3. 运行模式

伺服驱动器按照所设定的运行模式运行, 设备可以实现多种运行模式。推荐伺服驱动器实现的运行模式如表 5.16 所列。主站通过写数据对象 0x6060 来设定运行模式, 从站驱动设备用数据对象 0x6061 表示实际运行模式。0x6060 和 0x6061 的数据类型都是字节型。

表 5.16 CiA 402 运行模式定义

编 码	运行模式	缩 写	分 类	备 注
-128~-1	制造商定义运行模式			
0	没有分配运行模式			
1	定位控制模式	pp (profile positon)	O	
2	速度模式	vl (velocity)	O	变频器控制
3	升降速控制模式	pv (profile Velocity)	O	
4	扭矩模式	tq (torque)	O	
5	保留			
6	回零模式	hm (homing)	C	支持回零功能时必备
7	插补位控模式	ip (interpolation position)	O	
8	周期性同步位置模式	csp (cyclic synchronous position)	C	支持位控功能时必备
9	周期性同步速度模式	csv (cyclic synchronous velocity)	C	支持速度功能时必备
10	周期性同步扭矩模式	cst (cyclic synchronous torque)	C	支持扭矩功能时必备
11~127	保留			

数据对象 0x6062 表示驱动设备支持的运行模式, 它按位定义, 每一位对应一种运行模式,

如图 5.13 所示。

31	16 15	10 9	8	7	6	5	4	3	2	1	0
制造商定义	保留	cst	csv	csp	ip	hm	r	tq	pv	vl	pp

图 5.13 数据对象 0x6062 定义

其中周期性同步运行模式是 CoE 对 CiA 402 的扩展，在数控设备中得到广泛应用。现对其进行介绍。

(1) 周期性同步位置模式 csp (cyclic synchronous position)

周期性同步位置控制模式的结构如图 5.14 所示，位置指令由控制主站生成，它向驱动设备发送周期性同步的位置指令值。驱动设备执行位置控制、速度控制和扭矩控制。这样多个伺服驱动装置可以严格地同步进行协调运动，实现精密的轮廓轨迹控制。另外，控制主站也可以提供附加的速度和扭矩值，实现速度和扭矩前馈控制。驱动设备可以向控制设备提供实际位置值、实际速度值和实际扭矩值。伺服设备也监测跟随误差、实现速度限制和急停功能。

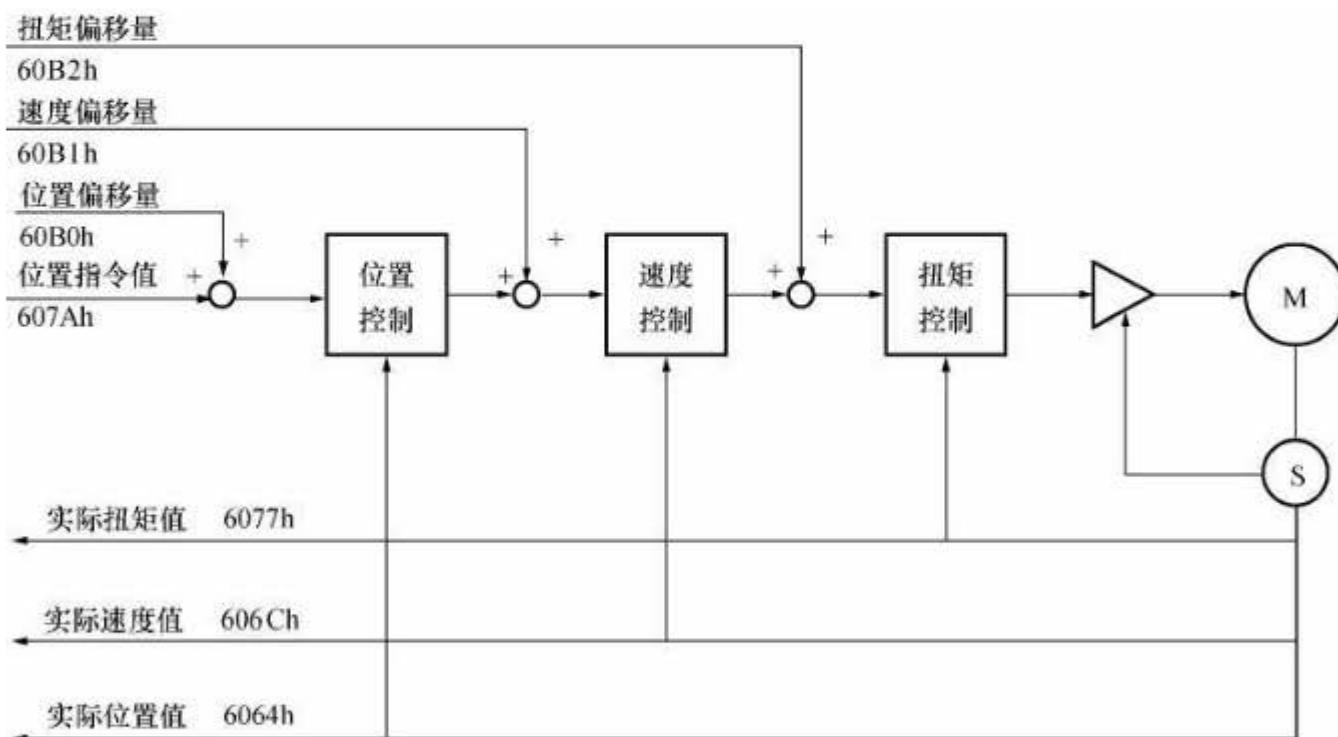


图 5.14 周期性同步位置控制运行模式结构图

(2) 周期性同步速度模式 csv (cyclic synchronous velocity)

周期性同步速度控制模式结构如图 5.15 所示，控制主站周期性地向驱动设备发送目标速度指令。驱动设备进行速度控制和扭矩控制，如果需要的话，位置环可以通过控制主站而闭合。控制主站也可以提供附加扭矩值，实现扭矩前馈控制。驱动设备可以向控制主站提供实际位置值、实际速度值和实际扭矩值。

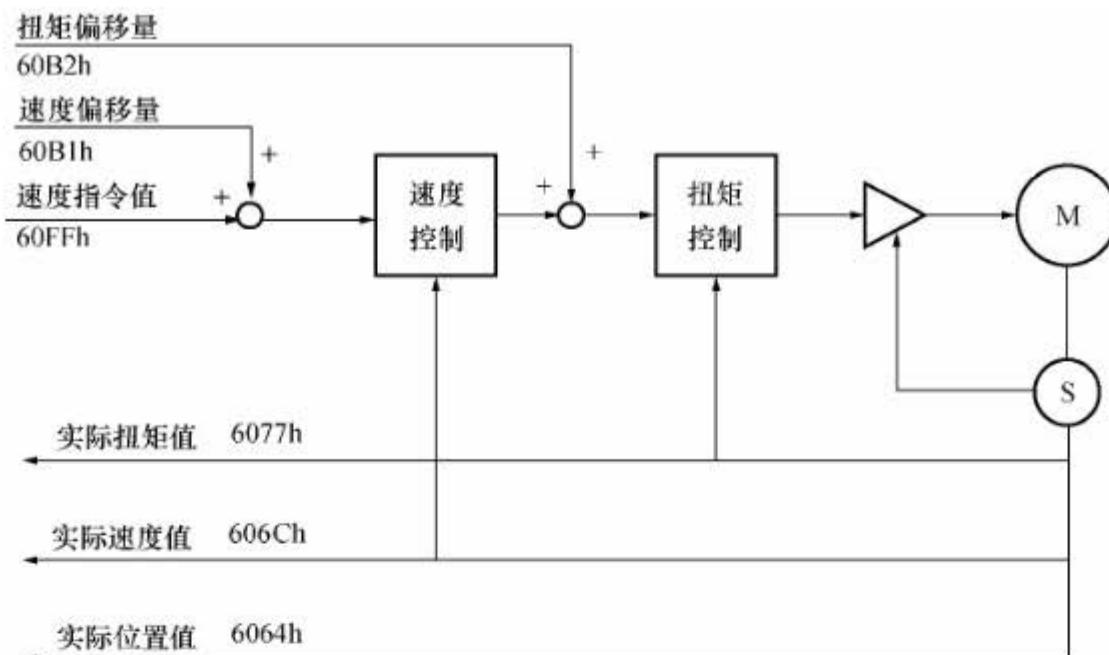


图 5.15 周期性同步速度控制运行模式结构图

(3) 周期性同步扭矩控制模式 cst (cyclic synchronous torque)

周期性同步扭矩控制运行模式结构如图 5.16 所示。控制主站周期性地向驱动设备发送目标扭矩指令，驱动设备运行扭矩控制。驱动设备可以向控制主站提供实际位置值、实际速度值和实际扭矩值。

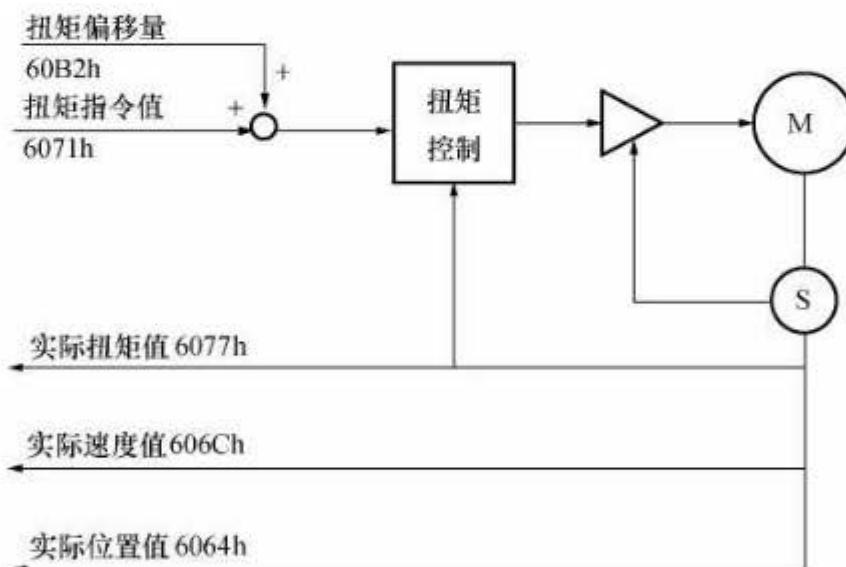


图 5.16 周期性同步扭矩控制运行模式结构图

为了在运行状态下动态切换运行模式，将位置指令、速度指令和扭矩指令全部映射到 PDO 数据中，同时并将运行模式 $0x6060$ 和 $0x6061$ 也映射到 PDO 数据中，主站和从站按照实际运行模式选用 PDO 中的映射数据（如表 5.17 所列）。一次运行模式切换的步骤如下：

- 初始化完成后，运行于某一模式下，主站负责更新所有与当前运行模式相关的过程数据；
- 如果主站选择了一种新的运行模式，RxPDO 中的运行模式数据对象 $0x6060$ 为新选用的模式代码；
- 从站收到 RxPDO 数据，发现运行模式发生改变后，进行内部模式切换，这需要一定时间，此时仍反馈旧模式下的过程数据信息；
- 主站在此中间状态下必须同时发送旧运行模式和新运行模式的有效数据，直到从站

反馈的实际运行模式等于主站设置的运行模式；

- 控制字中的模式相关位必须与 RxPDO 中的运行模式保持一致，包括处于中间状态时；
- 状态字中的模式相关位必须与 TxPDO 中的实际运行模式保持一致，包括处于中间状态时。

表 5.17 动态切换运行模式时的 PDO 映射

PDO	映射数据	含 义	数据长度
RXPDO 映射参数 0x1704 主站发送给从站的指令数据	0x607A:00	位置指令值	4 字节
	0x60FF:00	速度指令值	4 字节
	0x6071:00	扭矩指令值	2 字节
	0x6060:00	运行模式	1 字节
	0x6040:00	控制字	2 字节
TXPDO 映射参数 0x1B08 从站发送给主站的反馈数据	0x6064:00	实际位置值	4 字节
	0x6077:00	实际扭矩值	2 字节
	0x60F4:00	跟随误差	4 字节
	0x6061:00	实际运行模式	1 字节
	0x6041:00	状态字	2 字节

5.2 SoE (SERCOS over EtherCAT)

SERCOS 接口是一种高性能的数字伺服实时通信接口协议，于 1995 年被批准为国际标准 IEC61491。它包括通信技术和多种设备行规。SoE 是指在 EtherCAT 协议下运行 SERCOS 协议定义的伺服设备行规，使用 EtherCAT 通信网络和协议操作 SERCOS 设备行规定义的伺服参数和控制数据，EtherCAT 通信网络不传输 SERCOS 接口链路层协议。SoE 协议允许在 EtherCAT 环境中集成基于 SERCOS 设备行规的设备，包括 SERCOS 状态机（通信阶段）、同步、过程数据通信和通过服务通道访问 IDN 参数。关于详细的 SERCOS 接口协议请参考作者的另外一本专著《数字伺服通信协议 SERCOS 驱动程序设计及应用》^[15]。SoE 协议的内容主要包括：

- (1) EtherCAT 状态机与 SERCOS 通信阶段的对应；
- (2) SoE 对 SERCOS 协议 IDN 参数的继承；
- (3) SERCOS 周期性数据报文 MDT (Master Data Telegram) 和 AT (Driver Telegram) 与 EtherCAT 周期性数据帧传输的对应；
- (4) 取消 MST，由 EtherCAT 分布时钟实现精确同步；
- (5) SERCOS 服务通道与 EtherCAT 邮箱通信的对应，实现 IDN 访问操作。

5.2.1 SoE 状态机

SERCOS 协议的通信阶段与 EtherCAT 状态机的比较如图 5.17 所示，其特点有以下几个方面：

- ① SERCOS 协议通信阶段 0 和 1 被 EtherCAT 初始化状态覆盖；

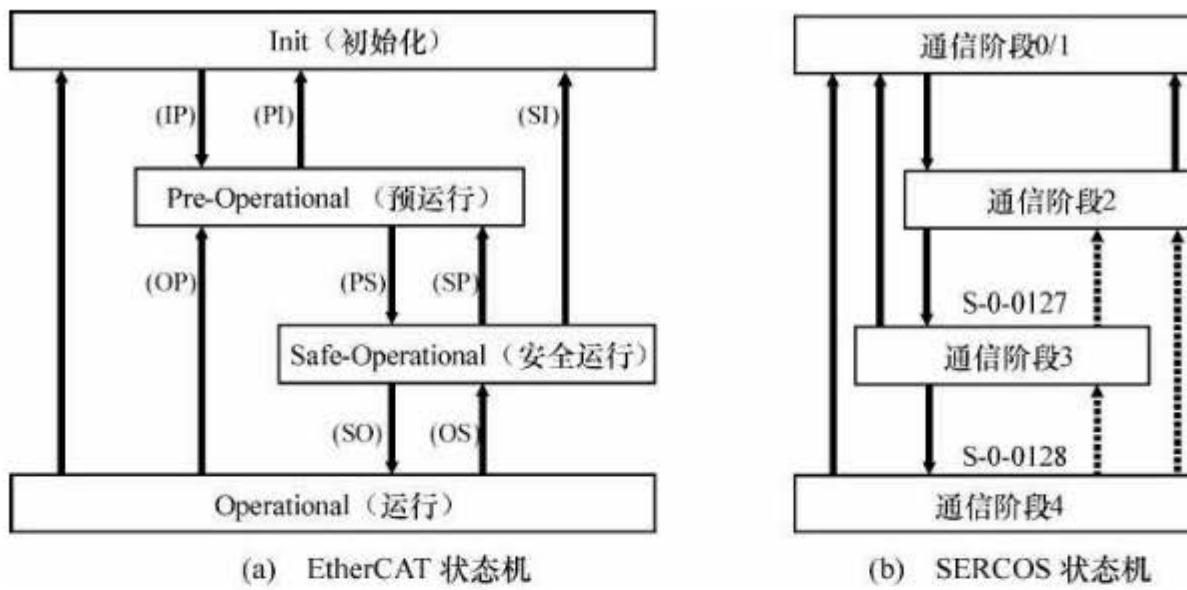


图 5.17 EtherCAT 状态机与 SERCOS 状态机比较

- ② 通信阶段 2 对应于预运行状态，允许使用邮箱通信实现服务通道，操作 IDN 参数；
- ③ 通信阶段 3 对应于安全运行状态，开始传输周期性数据，只有输入数据有效，输出数据被忽略，同时可以实现时钟同步；
- ④ 通信阶段 4 对应于运行阶段，所有的输入和输出都有效；
- ⑤ 不使用 SERCOS 协议的阶段切换过程命令 S-0-0127(通信阶段 3 切换检查)和 S-0-0128(通信阶段 4 切换检查)，分别由 PS 和 SO 状态转化取代；
- ⑥ SERCOS 协议只允许高级通信阶段向下切换到通信阶段 0，而 EtherCAT 允许任意的状态向下切换（如图 5.17(a)所示）。例如从运行状态切换到安全运行状态，或从安全运行状态切换到预运行状态。SoE 也应该支持这种切换（如图 5.17 (b) 中虚线所示），如果从站不支持，则应该在 EtherCAT AL 状态寄存器中设置错误位。

5.2.2 IDN 继承

SoE 协议继承 SERCOS 协议的 IDN 参数定义。每个 IDN 参数都有一个唯一的 16 位标识号 IDN，对应一个唯一的数据块，保存参数的全部信息。数据块由 7 个元素组成，如表 5.18 所列。IDN 参数分为标准数据和产品数据两部分，每部分又分为 8 个参数组，使用不同的 IDN 表示，如表 5.19 所列。

在使用 EtherCAT 作为通信网络时，取消了一些在 SERCOS 协议中用于通信接口控制的 IDN，如表 5.20 所列。此外，还对一些 IDN 的定义做了修改，如表 5.21 所列。

表 5.18 IDN 数据块结构

编 号	名 称	备 注
元素 1	IDN	必备注
元素 2	名称	可选
元素 3	属性	必备注
元素 4	单位	可选
元素 5	最小允许值	可选
元素 6	最大允许值	可选
元素 7	数据值	必备注

表 5.19 IDN 编号定义

位	15	14~12	11~0
含义	分类	参数组	数据编号
取值	0: 标准数据 S 1: 产品数据 P	0~7: 8 个参数组	0000~4095

表 5.20 取消的 IDN

IDN	描述
S-0-0003	最小 AT 发送的开始时间 ($T_{1\min}$)
S-0-0004	发送到接收状态切换时间 (TATMT)
S-0-0005	最小反馈采样提前时间 ($T_{4\min}$)
S-0-0009	主站数据报文中的开始地址 (MDT POS)
S-0-0010	主站数据电报长度 (MDT LEN)
S-0-0088	接收 MDT 后准备好接收 MST 所需要的恢复时间 (TMTSG)
S-0-0090	命令值处理时间 (TMTSG)
S-0-0127	通信阶段 3 切换检查, 由 EtherCAT “PS” 状态切换替代, 如果切换失败, 失败原因保存在 IDN S-0-0021
S-0-0128	通信阶段 4 切换检查, 由 EtherCAT “SO” 状态切换替代, 如果切换失败, 失败原因保存在 IDN S-0-0022

表 5.21 修改定义的 IDN

IDN	原功能	新功能
S-0-0006	AT 发送开始时间 (T_1)	在从站内部于同步信号之后应用程序向 ESC 存储区写入 AT 数据的时间偏移
S-0-0014	通信接口状态	映射从站 DL 状态、AL 状态和 AL 状态码
S-0-0028	MST 错误计数	映射从站 RX 错误计数器和连接丢失计数器
S-0-0089	MDT 发送开始时间 (T_2)	在从站内部于同步信号之后从 ESC 存储区得到新的 MDT 数据的时间偏移

5.2.3 SoE 过程数据映射

输出过程数据 (MDT 数据内容) 和输入过程数据 (AT 数据内容) 由 S-0-0015、S-0-0016 和 S-0-0024 配置。过程数据不包括服务通道数据, 只有周期性过程数据。输出过程数据包括伺服控制字和指令数据, 输入过程包括状态字和反馈数据。S-0-0015 设定了周期性过程数据的类型, 如表 5.22 所列。主站在“预运行”阶段通过邮箱通信写这三个参数, 以配置周期性过程数据的内容。

表 5.22 参数 S-0-0015 定义

S-0-0015	指令数据	反馈数据
0: 标准类型 0	无指令数据	无反馈数据
1: 标准类型 1	扭矩指令值 S-0-0080 (2 字节)	无反馈数据
2: 标准类型 2	速度指令值 S-0-0036 (4 字节)	速度反馈值 S-0-0040 (4 字节)
3: 标准类型 3	速度指令值 S-0-0036 (4 字节)	位置反馈值 S-0-0051 (4 字节)
4: 标准类型 4	位置指令值 S-0-0047 (4 字节)	或位置反馈值 S-0-0053 (4 字节)
5: 标准类型 5	位置指令值 S-0-0047 (4 字节) + 速度指令值 S-0-0036 (4 字节)	位置反馈值 S-0-0051 (4 字节) 或位置反馈值 S-0-0053 (4 字节) + 速度反馈值 S-0-0040 (4 字节)
6: 标准类型 6	速度指令值 S-0-0036 (4 字节)	无反馈数据
7: 自定义	S-0-0024 配置	S-0-0016 配置

输出过程数据映射如图 5.18 所示，其指令数据由 S-0-0024 配置，S-0-0024 定义如表 5.23 所列。控制字定义如表 5.24 所列。

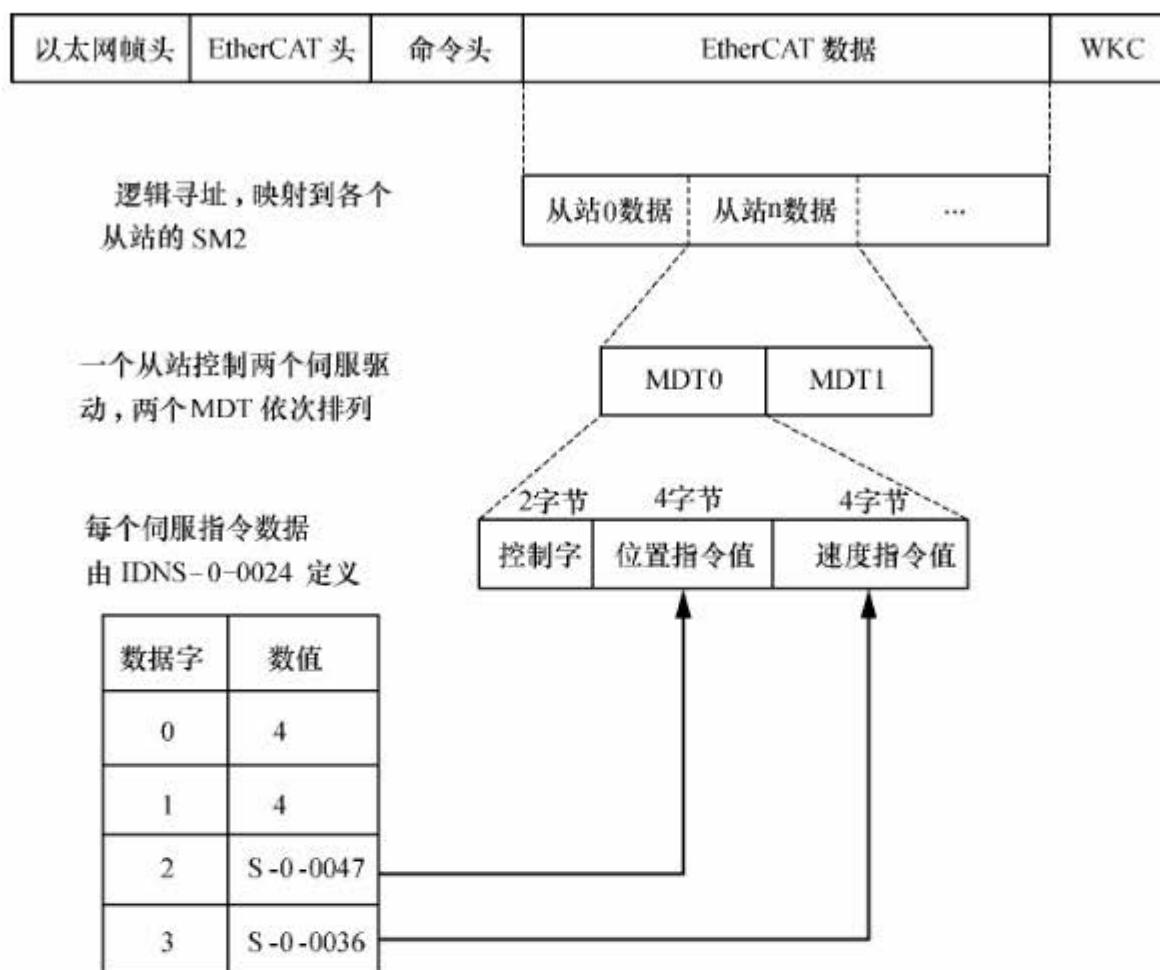


图 5.18 主站输出过程数据映射

表 5.23 S-0-0016 和 S-0-0024 定义

数据字	S-0-0024 定义	S-0-0016 定义
0	输出数据最大长度 (Word)	输入数据最大长度 (Word)
1	输出数据实际长度 (Word)	输入数据实际长度 (Word)
2	指令数据映射的第一个 IDN	反馈数据映射的第一个 IDN
3	指令数据映射的第二个 IDN	反馈数据映射的第二个 IDN
...

表 5.24 伺服控制字定义

位	描述
位 15	伺服动力电通/断 0: 伺服动力电断开: 当从“1”变为“0”时, 伺服以最优方式减速, 在速度达到 n_{min} 时停止扭矩输出 1: 伺服动力电接通
位 14	伺服环启/停 0: 伺服退出: 当从“1”变为“0”时, 停止扭矩输出 (独立于位 15 和位 13) 1: 伺服就绪, 进入正常工作状态

续表 5.24

位	描述
位 13	当位 15 和位 14 均为“1”时，暂停/重新启动伺服 0：暂停伺服：当从“1”变为“0”时，伺服驱动器停止，控制环保持闭合 1：重新启动伺服：当从“0”变为“1”时，伺服按加速度参数执行当前功能
位 12	保 留
位 10	控制单元同步位，初始值为 0。在周期性运行阶段有效，随着控制单元通信周期而翻转，表示命令数据值的更新
位 11,9,8	运行模式选择 0 0 0：主运行模式，由 IDN S-0-0032 定义 0 0 1：辅助运行模式 1，由 IDN S-0-0033 定义 0 1 0：辅助运行模式 2，由 IDN S-0-0034 定义 0 1 1：辅助运行模式 3，由 IDN S-0-0035 定义 1 0 0：辅助运行模式 4，由 IDN S-0-0284 定义 1 0 1：辅助运行模式 5，由 IDN S-0-0285 定义 1 1 0：辅助运行模式 6，由 IDN S-0-0286 定义 1 1 1：辅助运行模式 7，由 IDN S-0-0287 定义
位 7	实时控制功能位 2，由 S-0-0302 定义
位 6	实时控制功能位 1，由 S-0-0300 定义
位 5,4,3,2,1	保 留
位 0	翻 转

输入过程数据如图 5.19 所示，其反馈数据由 S-0-0016（AT 配置列表）配置，S-0-0016 定义如表 5.23 所列。从站伺服状态字定义如表 5.25 所列。

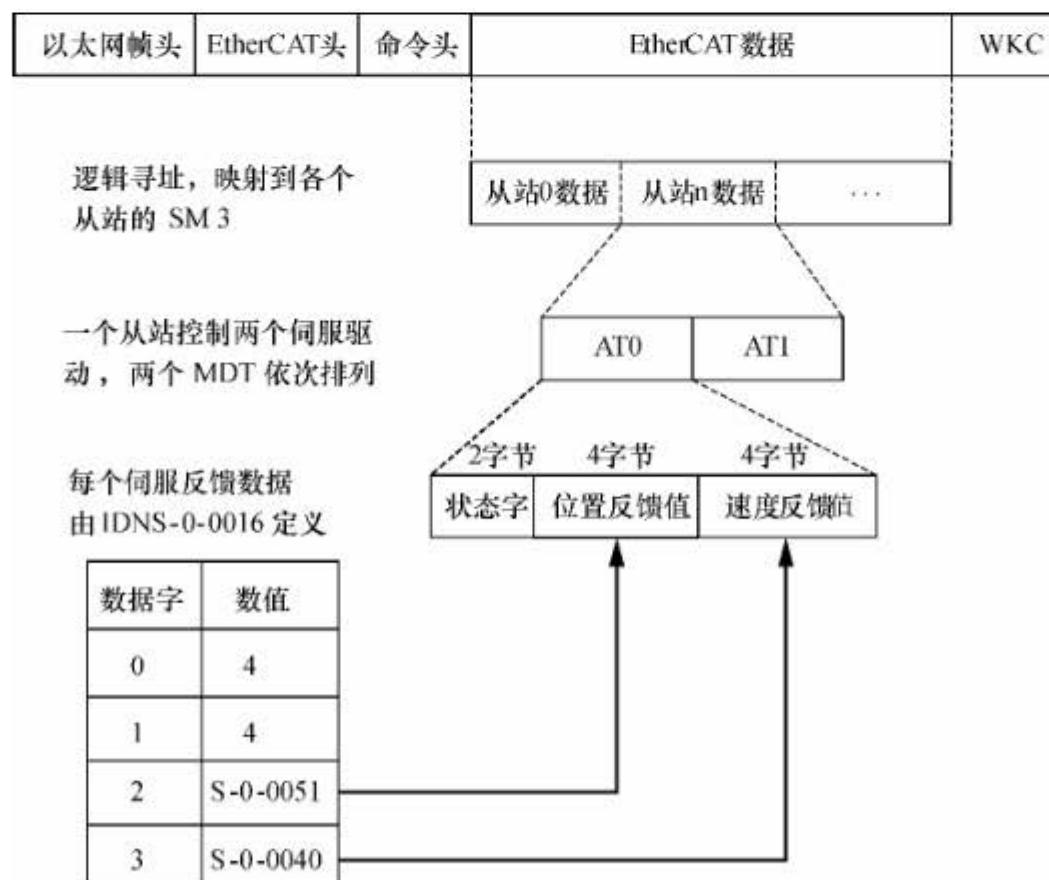


图 5.19 从站反馈数据映射

表 5.25 从站伺服状态字定义

位	描述
位 15, 14	伺服驱动器就绪 0 0: 伺服尚未准备好电源供电, 内部检测尚未顺利结束 0 1: 伺服已经准备好电源供电 1 0: 伺服电源就绪, 主电源供电, 伺服无扭矩输出 1 1: 伺服运行准备就绪, 伺服有扭矩输出
位 13	第 1 诊断类故障标志, 故障信息在参数 IDN 00011 中 0: 无故障 1: 发生第 1 诊断类故障, 伺服关闭
位 12	第 2 诊断类故障标志, 故障信息在参数 IDN 00012 中 0: 无变化 1: 有变化
位 11	第 3 诊断类故障标志, 故障信息在参数 IDN 00012 中 0: 无变化 1: 有变化
位 10,9,8	实际运行模式 0 0 0: 主运行模式, 由 IDN S-0-0032 定义 0 0 1: 辅助运行模式 1, 由 IDN S-0-0033 定义 0 1 0: 辅助运行模式 2, 由 IDN S-0-0034 定义 0 1 1: 辅助运行模式 3, 由 IDN S-0-0035 定义 1 0 0: 辅助运行模式 4, 由 IDN S-0-0284 定义 1 0 1: 辅助运行模式 5, 由 IDN S-0-0285 定义 1 1 0: 辅助运行模式 6, 由 IDN S-0-0286 定义 1 1 1: 辅助运行模式 7, 由 IDN S-0-0287 定义
位 7	实时状态位 2, 由 S-0-0306 定义
位 6	实时状态位 1, 由 S-0-0304 定义
位 5,4	保留
位 3	命令值处理状态 0: 伺服驱动器放弃了命令数据值 1: 伺服驱动器跟随命令数据值
位 2,1	保留
位 0	翻转位

5.2.4 SoE 服务通道

EtherCAT SoE 服务通道 SSC (SoE Service Channel) 由 EtherCAT 邮箱通信实现, 它用于非周期性数据交换, 如读写 IDN 及其元素。SoE 数据格式如图 5.20 所示。邮箱数据头之后是 4 个字节的 SoE 数据头, 它定义了 SoE 操作模式和所操作的 IDN 及其元素, 如表 5.26 所列。

其中操作元素标识定义如表 5.18 所列。

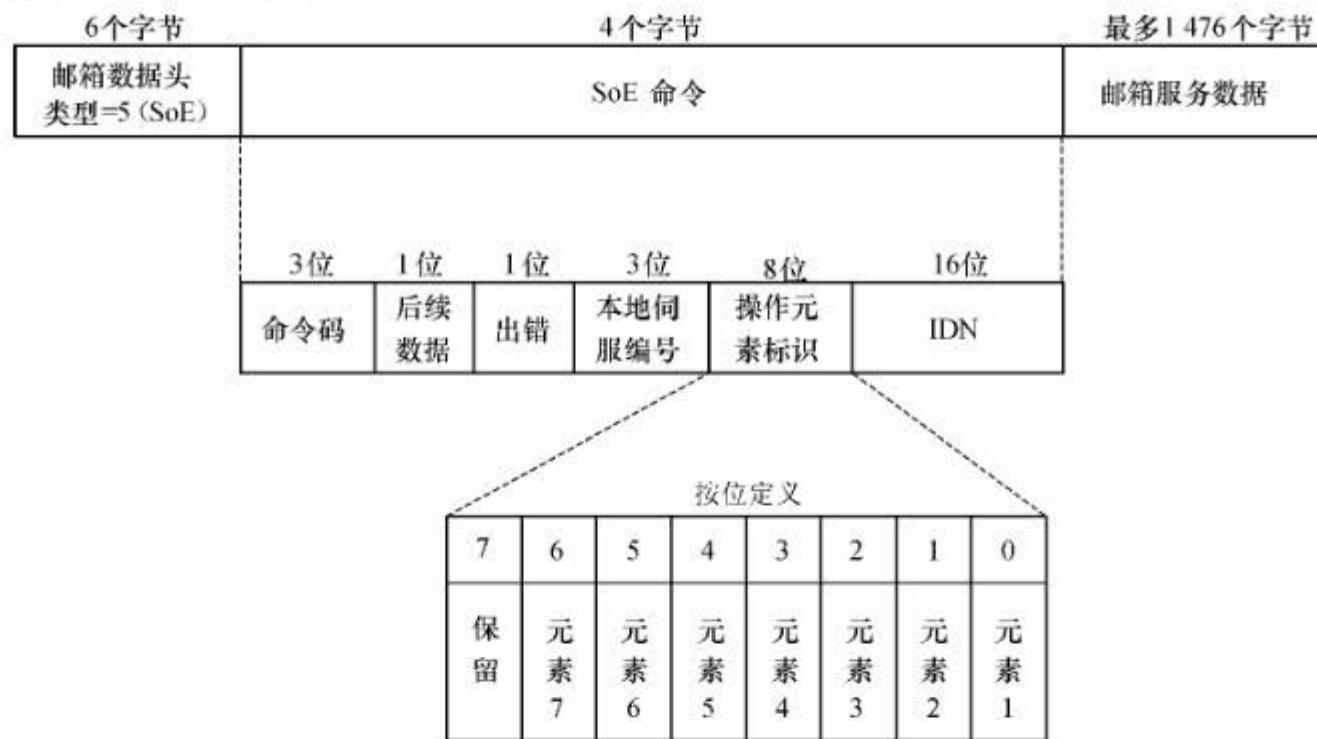


图 5.20 SoE 数据格式

表 5.26 SoE 邮箱协议

数据区	字节数	位 数	名 称	取值和描述
邮箱头	2 字节	16 位	长度 n	后续邮箱服务数据长度
	2 字节	16 位	地址	
	1 字节	位 0~5	通道	0x00: 保留
		位 6~7	优先级	0x00: 最低优先级 ... 0x03: 最高优先级
	1 字节	位 0~3	类型	0x05: SoE
		位 4~7	保留	0x00
SoE 数据头	1 字节	位 0~2	命令码 (OpCode)	0x01: 读请求 0x02: 读响应 0x03: 写请求 0x04: 写响应 0x05: 通报 0x06: 从站信息 0x07: 保留
				0x00: 无后续数据帧 0x01: 未完成传输, 有后续数据帧
				0x00: 无错误 0x01: 发生错误, 数据区有 2 个字节的错误码
				0x00: 无错误 0x01: 发生错误, 数据区有 2 个字节的错误码
				0x00: 无错误 0x01: 发生错误, 数据区有 2 个字节的错误码
				0x00: 无错误 0x01: 发生错误, 数据区有 2 个字节的错误码

续表 5.26

数据区	字节数	位数	名称	取值和描述
SOE 数据头	1 字节	位 5~7	伺服编号 (DroveNo)	从站本地伺服编号
			操作元素标识 (ElementFlags)	<ul style="list-style-type: none"> 单个元素操作时为元素选择，按位定义，每一位对应一个元素； 寻址结构体时为元素的数目
	2 字节	16 位	IDN	参数的 IDN 编号，或分段操作时的剩余片段
SoE 数据	(n-4) 字节		有效数据 或错误码	SoE 服务通道有效数据 发生错误时为 2 个字节的错误码

(1) SSC 读操作

SSC 读操作由主站发起，写 SSC 读请求到从站。从站收到读操作请求后，用所请求的参数 IDN 编号和数据值作为回答，如图 5.21 所示。主站可以同时读多个元素，从站应该同时回答多个元素，如果从站只支持单个元素操作，应该以所请求的第一个元素作为响应。

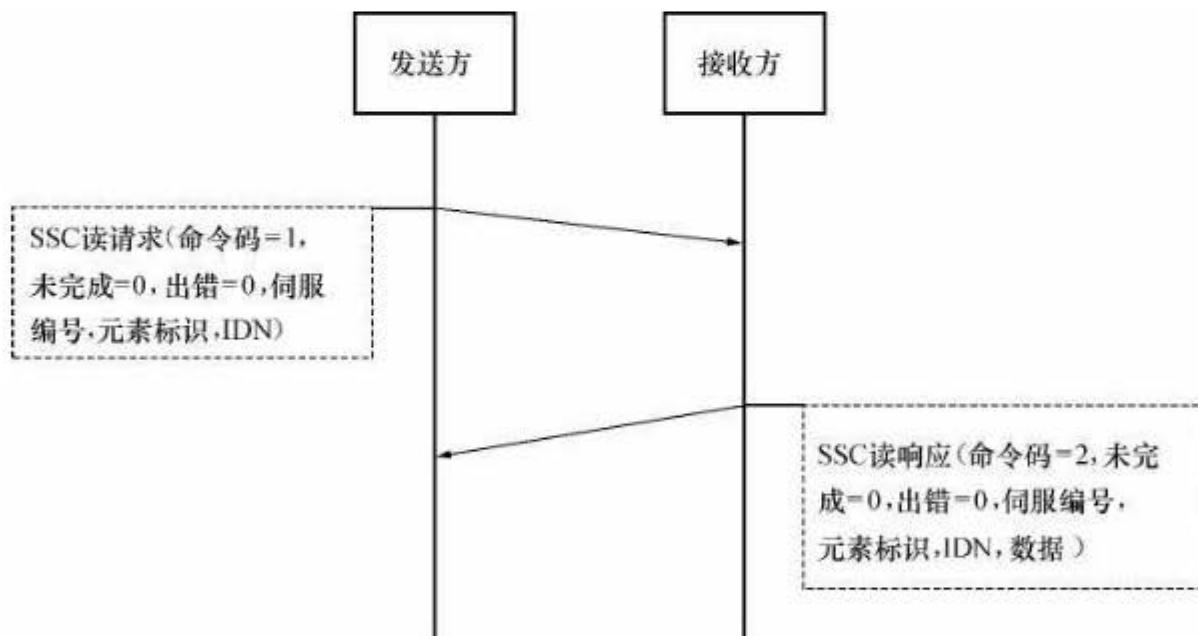


图 5.21 正确的 SSC 读操作序列

如果所需要读的数据长度超过了邮箱容量，则必须使用分段读操作。分段读操作由一个 SSC 读服务请求、一个或多个 SSC 分段读服务响应和一个 SSC 读服务响应组成，如图 5.22 所示。从站的分段读响应中若“未完成=1”，表示还有后续数据，此时用 IDN 域表示后续数据的片段数。

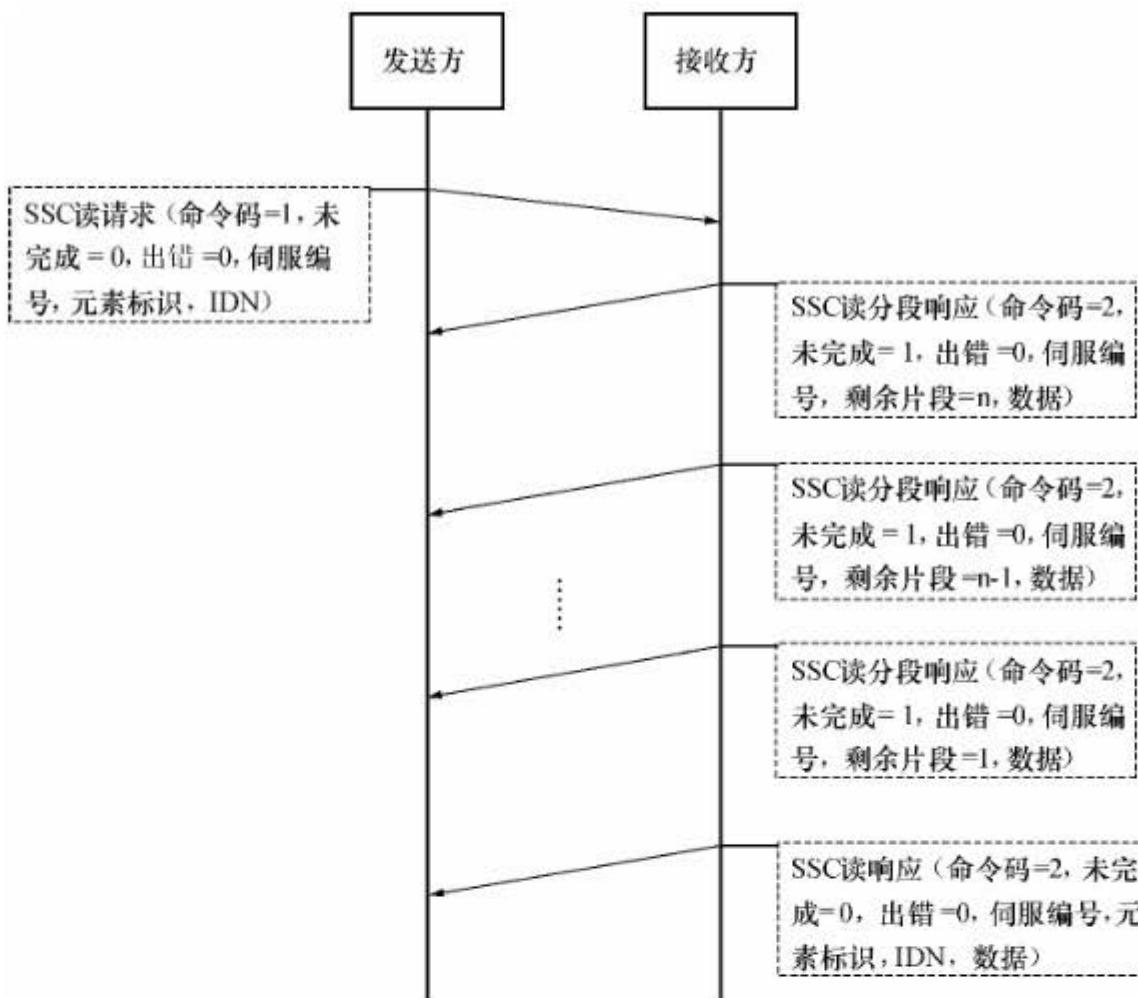


图 5.22 正确的 SSC 分段读操作序列

(2) SSC 写操作

SSC 写操作用于主站下载数据到从站，从站应该以写操作的结果回答。如图 5.23 所示。

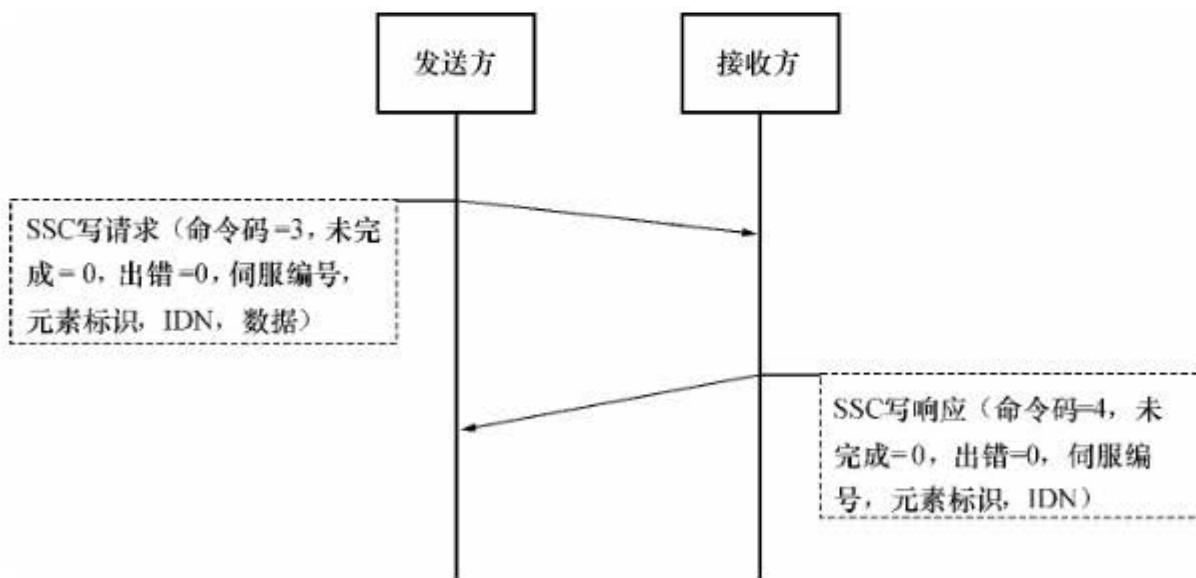


图 5.23 SSC 写操作时序

如果需要下载的数据超出邮箱容量，需要使用分段写操作。分段写操作由一个或多个分段写操作及一个 SSC 写响应服务组成，如图 5.24 所示。

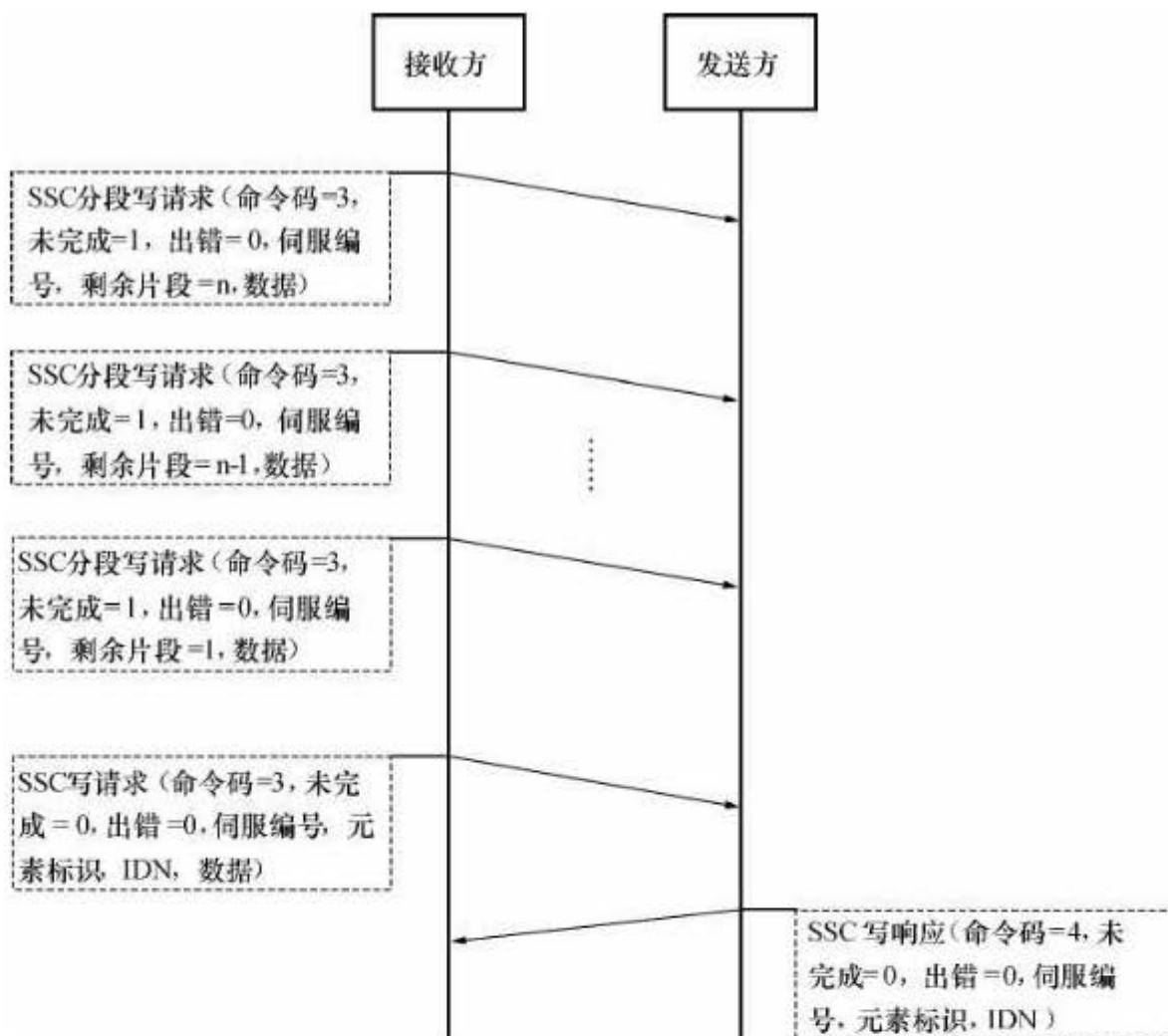


图 5.24 SSC 分段写操作时序

(3) SSC 过程命令

过程命令是一种特殊的非周期数据，每一个过程命令都有唯一的标识号 IDN 和规定的数据元素，用于启动伺服装置的某些特定功能或过程。执行这些功能或过程通常需要一段时间，过程命令只是触发其开始，随后它所占用的服务通道立即变为可用，用以传输其他非周期数据或过程命令，而不用等到被触发的功能或过程必须执行完毕。最常用的过程命令有“伺服装置控制的回原点过程命令 S-0-148”。

过程命令功能由主站启动，由从站执行。通过写过程命令 IDN 的元素 7(如表 5.18 所列)，将“过程命令控制”发往伺服装置，用以控制过程命令的设置、启动、中断和撤消。过程命令控制的数据格式如图 5.25 所示。

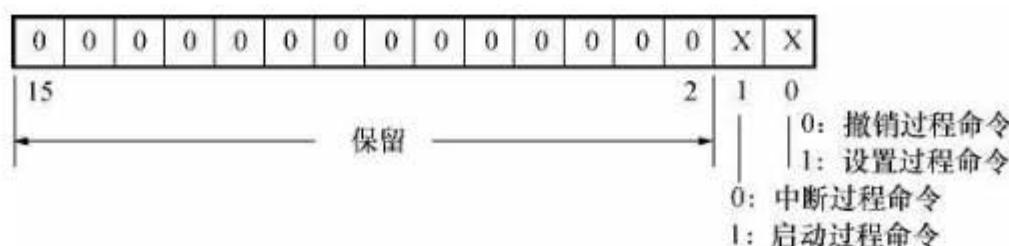


图 5.25 过程命令控制字

主站通过读过程命令 IDN 的元素 7 得到从站的过程命令状态字，过程命令状态字数据格式如图 5.26 所示。

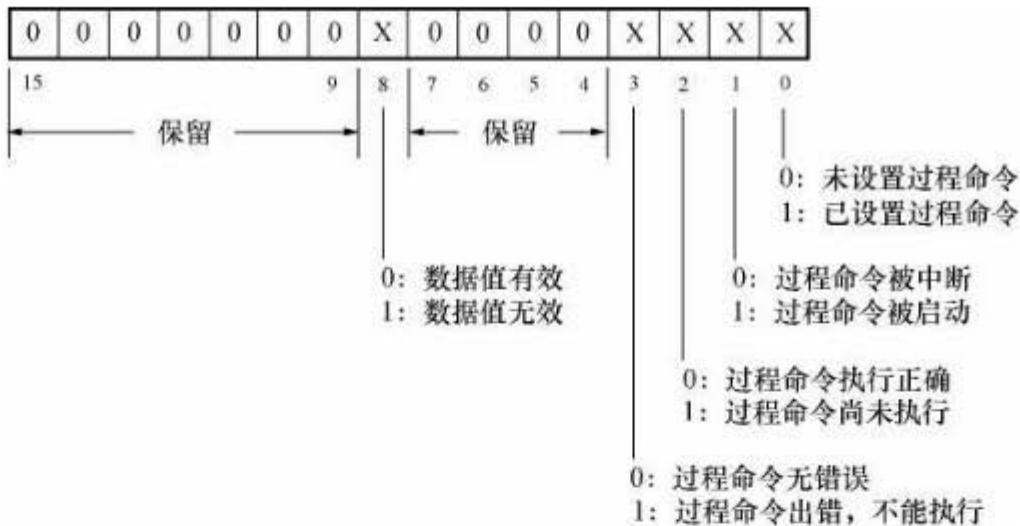


图 5.26 过程命令状态字

作为一个基本原则，每一个过程命令被处理以后，无论是获得执行正确应答，还是出现错误应答，主站都应该撤消该过程命令。具体方法是：将过程命令控制字的位 0 置为“0”后发往相应的伺服装置。

过程命令由 SSC 写一个特定的 IDN 发起。在过程命令功能启动之后，从站产生一个普通的 SSC 写响应数据。此时，服务通道空闲，又可以用于传输其他非周期性数据或更多过程命令。从站在执行完过程命令之后，发起一个 SSC 通报命令，SoE 数据头中命令码等于 5（如表 5.26 所列）。主站在读到从站通报命令之后，发出撤销过程命令请求，并读取过程命令状态，直到过程命令已被撤销，其具体执行流程如图 5.27 所示。

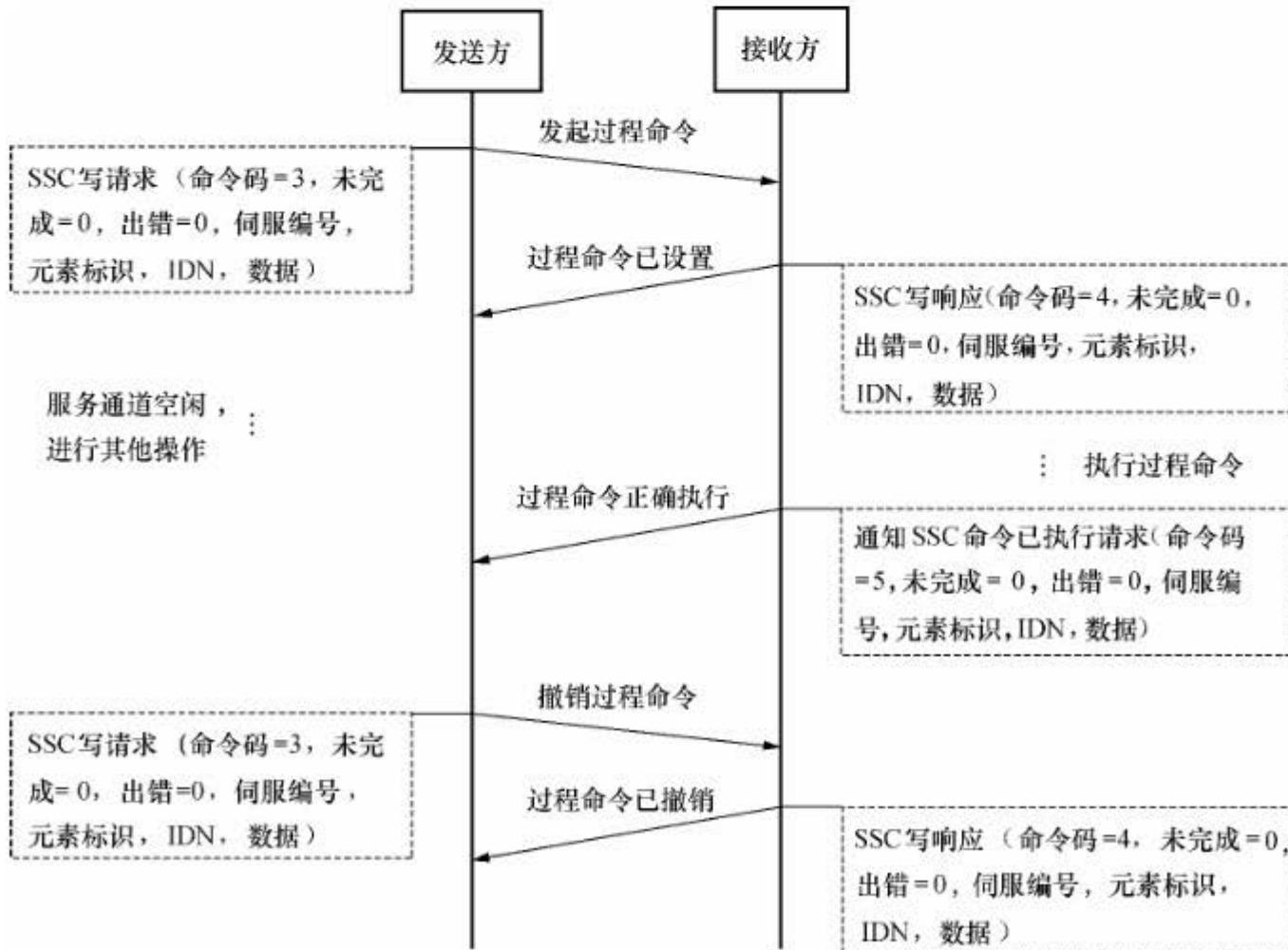


图 5.27 过程命令执行流程

主站也可以在过程命令执行过程中发送撤销过程命令控制字以终止过程命令。但不能终止一次非周期数据（包括参数和过程命令）的传输过程。

(4) SSC 从站信息服务

从站信息服务的主要目的是提供从站的附加信息便于系统调试和维护。从站信息服务由从站发起，将信息数据写入输入邮箱 SM1，由主站读取，如图 5.28 所示。

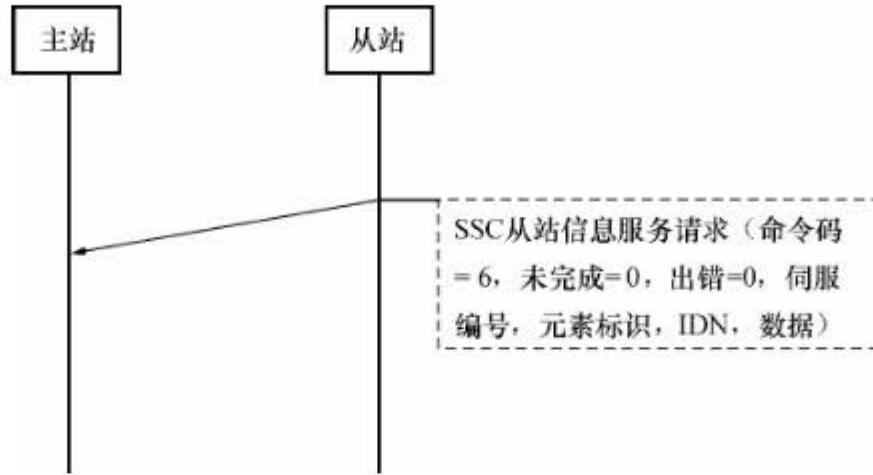


图 5.28 SSC 从站信息服务执行时序

第 6 章 EtherCAT 主站驱动程序

EtherCAT 主站可由 PC 计算机或其他嵌入式计算机实现，使用 PC 计算机构成 EtherCAT 主站时，通常用标准的以太网网卡 NIC (Network Interface Card) 作为主站硬件接口，主站功能由软件实现。从站使用专用芯片 ESC，通常需要一个微处理器实现应用层功能。EtherCAT 通信协议栈如图 6.1 所示。

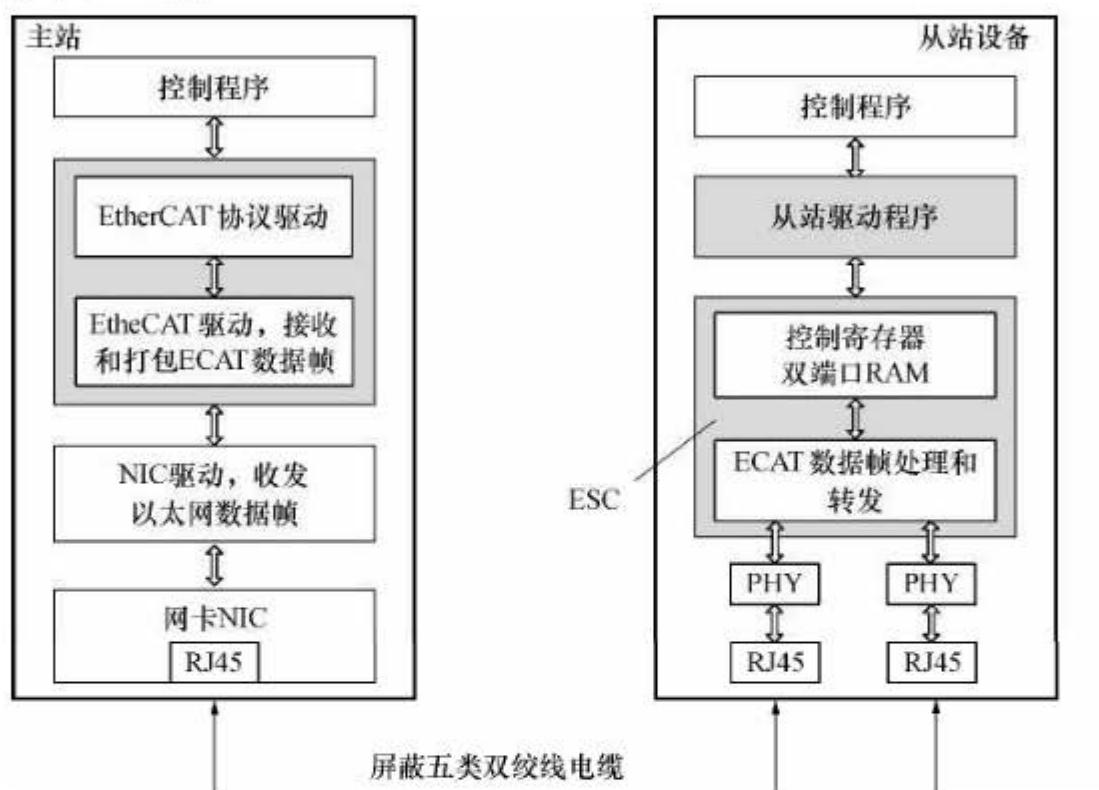


图 6.1 EtherCAT 控制系统协议栈

本章介绍作者在 Windows XP 操作系统下开发的 EtherCAT 主站驱动程序示例，实现基本的 EtherCAT 数据通信，包括 EtherCAT 通信初始化、周期性数据传输和非周期性数据传输。这些功能由以下三个基本类实现：

(1) 网卡操作相关类 CEcNpfDevice 和 CEcInfo

用以实现主站通信网卡的管理和 Ethernet 数据帧的收发。

(2) CEcSimSlave 类

定义了从站配置数据，构造一个从站对象。

(3) CEcSimMaster 类

用以实现主站的全部功能，包括网络初始化、周期性数据收发和非周期性数据收发。

本章首先介绍以上 3 个基本类，给出其关键程序流程图和部分程序源代码。接着介绍了一个 EtherCAT 主站应用程序实例。

6.1 数据定义头文件

EthernetService.h 头文件定义了以太网数据帧和 EtherCAT 数据帧相关的常量、数据结构和运算方法。以下为 Ethernet Service.h 头文件的程序源代码，其中一些数据将在后续章节中进一步介绍。

```

//=====
//EthernetService.h -- head file for Ethernet and EtherCAT frame
//=====

#ifndef _ENET_SERVICE_H_
#define _ENET_SERVICE_H_

// -----
#define ETHERNET_FRAME_TYPE_IP      0x0800 //IP数据帧的以太类型
#define ETHERNET_FRAME_TYPE_ECAT   0x88A4 //EtherCAT帧以太类型
#define ETHERNET_MAX_FRAME_LEN     1514 //以太网数据帧最大长度
//-----

// TETHERNET_ADDRESS: 以太网MAC地址数据结构定义
//-----

typedef struct TETHERNET_ADDRESS
{
    BYTE b[6];

#ifdef __cplusplus// 在C++编译环境中有效
    // 运算符“%”重载, 计算哈希值
    int operator% (int hashSize)
        { return (*((DWORD*)&b[2])) % hashSize; }

    // 运算符“==”重载, 和另一个MAC地址addr相比较
    BOOL operator== ( const TETHERNET_ADDRESS &addr ) const
        { return ( memcmp( b, addr.b, sizeof(ETHERNET_ADDRESS) ) == 0 ); }

    // 运算符“!=”重载, 和另一个MAC地址addr相比较
    BOOL operator!= ( const TETHERNET_ADDRESS &addr ) const
        { return ( memcmp( b, addr.b, sizeof(ETHERNET_ADDRESS) ) != 0 ); } // 和另一个MAC地址addr相比较

    // 运算符“=”重载, 用另一个Mac地址给本MAC地址赋值
    void operator= ( const TETHERNET_ADDRESS &addr )
        { memcpy( &b, &addr.b, sizeof(b) ); }

    // 判断是否为有效MAC地址
    BOOL IsValid()
        { return ((*((DWORD*)&b[0])) != 0 || (((USHORT*)&b[4])) != 0; }

```

```

// 清除MAC地址
void Clear()
{
    memset( &b, 0, sizeof(b) );
}

#ifndef IS_R0
// 将MAC地址转换为字符串，结果由参数调用返回
void Id2String( CString& rcsAddr )
{
    rcsAddr.Format( _T("%02x %02x %02x %02x %02x %02x"),
        b[0], b[1], b[2], b[3],
        b[4], b[5] );
}

// 将MAC地址转换为字符串，结果赋值给一个字符串
CString Id2String()
{
    CString sAddr;
    sAddr.Format( _T("%02x %02x %02x %02x %02x %02x"),
        b[0], b[1], b[2],
        b[3], b[4], b[5] );
    return sAddr;
}

// 运算符“=”重载，用一个字符串类给本MAC地址赋值
void operator= ( CString csAddr )
{
    operator= ( (LPCTSTR) csAddr );
}

// 运算符“=”重载，用一个字符串变量给本MAC地址赋值
void operator= ( LPCTSTR csAddr )
{
    int d[6], i;
    memset( b, 0, sizeof(b) );
    if ( csAddr[0] && _stscanf( csAddr, _T("%02x %02x %02x %02x
        %02x %02x"), &d[0], &d[1], &d[2], &d[3], &d[4], &d[5] )==6 )
    {
        for ( i = 0; i < 6; i++ )
            b[i] = d[i];
    }
}

#endif // !IS_R0
#endif // __cplusplus
} ETHERNET_ADDRESS, *PETHERNET_ADDRESS;
#define ETHERNET_ADDRESS_LEN sizeof(ETHERNET_ADDRESS)

//-----//
// 常用MAC地址定义

```

```

//-----  

//--广播地址-----  

const ETHERNET_ADDRESS BroadcastEthernetAddress =  

    {0xff,0xff,0xff,0xff,0xff,0xff};  

//--第一个多播地址-----  

const ETHERNET_ADDRESS FirstMulticastEthernetAddress =  

    {0x01,0,0x5e,0,0,0};  

//--空MAC地址-----  

const ETHERNET_ADDRESS NullEthernetAddress={0,0,0,0,0,0};

//-----  

// 以太网数据帧头数据结构定义  

//-----  

typedef struct TETHERNET_FRAME  

{  

    ETHERNET_ADDRESS Destination;           // 目的地地址  

    ETHERNET_ADDRESS Source;                // 源地址  

    USHORT           FrameType;             // 主机字节顺序  

} ETHERNET_FRAME, *PETHERNET_FRAME;  

#define ETHERNET_FRAME_LEN      sizeof(ETHERNET_FRAME)  

#define ETHERNET_FRAMETYPE_LEN  sizeof(USHORT)  

#define FRAMETYPE_PTR(p)  

    (((PUSHORT)p)[6]==ETHERNET_FRAME_TYPE_VLAN_SW ?  

     &((PUSHORT)p)[8] : &((PUSHORT)p)[6])  

#define ENDOF_ETHERNET_FRAME(p)  ENDOF(FRAMETYPE_PTR(p))

//-----  

// EtherCAT数据头定义  

//-----  

typedef struct ETTYPE_88A4_HEADER  

{  

    USHORT Length   : 11;           // 后续数据长度  

    USHORT Reserved : 1;            // 保留  

    USHORT Type     : 4;            // 由ETTYPE_88A4_TYPE_xxx定义  

} ETTYPE_88A4_HEADER, *PETTYPE_88A4_HEADER;  

#define ETTYPE_88A4_HEADER_LEN  sizeof(ETTYPE_88A4_HEADER)

//-----  

// EtherCAT数据帧类型定义

```

```

//-----  

#define ETYPE_88A4_TYPE_ECAT 1 // ECAT header follows  

#define ETYPE_88A4_TYPE_ADS 2 // ADS header follows  

#define ETYPE_88A4_TYPE_IO 3 // I/O  

#define ETYPE_88A4_TYPE_NV 4 // Network Variables  

#define ETYPE_88A4_TYPE_CANOPEN 5 // ETHERCAT_CANOPEN_HEADER  

                           // follows  

//-----  

// EtherCAT数据帧定义  

//-----  

typedef struct TETHERNET_88A4_FRAME  

{  

    ETHERNET_FRAME           Ether;  

    ETYPE_88A4_HEADER        E88A4;  

} ETHERNET_88A4_FRAME, *PETHERNET_88A4_FRAME;  

//--- EtherCAT数据帧头长度计算 ---  

#define ETHERNET_88A4_FRAME_LEN sizeof(ETHERNET_88A4_FRAME)  

//--- EtherCAT数据帧长度计算 ---  

#define SIZEOF_88A4_FRAME(p)  

    (sizeof(ETHERNET_88A4_FRAME)+((PETHERNET_88A4_FRAME)(p))->E8  

     A4.Length)  

//--- EtherCAT数据帧尾地址计算 ---  

#define ENDOF_88A4_FRAME(p)  

    ((PETHERNET_88A4_FRAME)&(((PBYTE)(p))[SIZEOF_88A4_FRAME(p)]))  

//-----  

// EtherCAT命令类型定义  

//-----  

typedef enum
{
    EC_CMD_TYPE_NOP = 0,  

    EC_CMD_TYPE_APRD= 1,  

    EC_CMD_TYPE_APWR= 2,  

    EC_CMD_TYPE_APRW= 3,  

    EC_CMD_TYPE_FPRD= 4,  

    EC_CMD_TYPE_FPWR= 5,  

    EC_CMD_TYPE_FPRW= 6,
}

```

```

    EC_CMD_TYPE_BRD= 7,
    EC_CMD_TYPE_BWR= 8,
    EC_CMD_TYPE_BRW= 9,
    EC_CMD_TYPE_LRD= 10,
    EC_CMD_TYPE_LWR= 11,
    EC_CMD_TYPE_LRW= 12,
    EC_CMD_TYPE_ARMW= 13,
    EC_CMD_TYPE_EXT = 255,
} EC_CMD_TYPE;

//-----  

// EtherCAT数据帧INDEX定义  

//-----  

#define EC_HEAD_IDX_ACYCLIC_MASK 0x80
#define EC_HEAD_IDX_SLAVECMD 0x80
#define EC_HEAD_IDX_EXTERN_VALUE 0xFF

//-----  

// EtherCAT命令数据结构定义  

//-----  

typedef struct TETYPE_EC_HEADER
{
    union
    {
        struct
        {
            BYTE cmd; // EtherCAT命令
            BYTE idx; // EtherCAT数据帧Index
        };
        USHORT cmdIdx;
    };
    union
    {
        struct
        {
            USHORTadp; // 站点地址
            USHORTado; // 站内偏移地址
        };
        ULONG laddr; // 逻辑地址
    };
};

```

```

union
{
    struct
    {
        USHORT len : 11; // 命令数据长度
        USHORT res : 4; // 保留
        USHORT next: 1; // 是否有后续命令标志
    };
    USHORT length;
};

USHORT irq;

} ETYPE_EC_HEADER, *PETYPE_EC_HEADER;

#define ETYPE_EC_HEADER_LEN sizeof(ETYPE_EC_HEADER)
#define ETYPE_EC_CNT_LEN sizeof(USHORT)
#define ETYPE_EC_OVERHEAD
    (ETYPE_EC_HEADER_LEN+ETYPE_EC_CNT_LEN)
#define ETYPE_EC_CMD_LEN(p)
    (ETYPE_EC_OVERHEAD+((PETYPE_EC_HEADER)p)->len)
#define ETYPE_EC_CMD_COUNTPTR(p)
    ((PUSHORT)&(((PBYTE)p)[(ETYPE_EC_HEADER_LEN+((PETYPE_EC_HEADER)p)->len)]))
#define ETYPE_EC_CMD_COUNT(p)
    (*((PUSHORT)&(((PBYTE)p)[(ETYPE_EC_HEADER_LEN+((PETYPE_EC_HEADER)p)->len)])))
#define ETYPE_EC_CMD_DATA(p)
    (*((PUSHORT)&(((PBYTE)p)[ETYPE_EC_HEADER_LEN])))
#define ETYPE_EC_CMD_DATAPTR(p)
    (&(((PBYTE)p)[ETYPE_EC_HEADER_LEN]))
#define NEXT_EcHeader(p)
    ((PETYPE_EC_HEADER)&((PBYTE)p)[((PETYPE_EC_HEADER)p)->len + ETYPE_EC_OVERHEAD])

//-----//
// EtherCAT最大数据结构定义
//-----//

typedef struct TETHERNET_88A4_MAX_HEADER
{
    ETYPE_88A4_HEADER      E88A4; // EtherCAT数据头
    union

```

```

{
    // EtherCAT第一个命令头
    struct
    {
        // ETTYPE_88A4_TYPE_ECAT
        ETTYPE_EC_HEADER FirstEcHead;
    };
    // EtherCAT数据
    BYTE Data
    [ETHERNET_MAX_FRAME_LEN-ETHERNET_88A4_FRAME_LEN];
};

} ETHERNET_88A4_MAX_HEADER, *PETHERNET_88A4_MAX_HEADER;

//-----//
// EtherCAT最长数据帧结构定义
//-----//

typedef struct TETHERNET_88A4_MAX_FRAME
{
    ETHERNET_FRAME Ether;          // 以太网数据帧头
    ETTYPE_88A4_HEADER E88A4;      // EtherCAT数据头
    union
    {
        // EtherCAT第一个命令头
        struct
        {
            // ETTYPE_88A4_TYPE_ECAT
            ETTYPE_EC_HEADER FirstEcHead;
        };
        // EtherCAT数据
        BYTEData
        [ETHERNET_MAX_FRAME_LEN-ETHERNET_88A4_FRAME_LEN];
    };
};

} ETHERNET_88A4_MAX_FRAME, *PETHERNET_88A4_MAX_FRAME;

//-----//
// 组织一个EtherCAT命令
//-----//

_inlineVOID FillEcHeaderAndData(PETYPE_EC_HEADER pHead,
BYTE cmd, BYTE idx, USHORT adp, USHORT ado, PVOID pData, ULONG nData,
BOOL next)
{
    PVOID pVoid;
}

```

```

pHead->cmd    = cmd;
pHead->idx    = idx;
pHead->adp    = adp;
pHead->ado    = ado;
pHead->len    = nData;
pHead->next   = next;
pHead->res    = 0;
pHead->irq    = 0;
pVoid = ENDOF(pHead);
if ( pData )
    memcpy(pVoid, pData, nData);
else
    memset(pVoid, 0, nData);
ETYPE_EC_CMD_COUNT(pHead) = 0;
}
#endif

```

6.2 网卡操作相关类的定义和实现

本例使用了一个开源的专业网络驱动开发包 WinPcap (Windows Packet Capture) 作为网卡驱动程序。它是 Libpcap (the Packet Capture Library) 在 Windows 平台下的版本，符合网络驱动程序接口规范 NDIS (Network Driver Interface Specification)。Libpcap 是一个平台独立的网络数据帧捕获开发包，由 Berkeley 大学的 Van Jacobson, Craig Ierес 和 Steven McCanne 编写，支持 Linux、Solaris 和 BSD 系统平台。WinPcap 主要由加利福尼亚大学的 Lawrence Berkeley Laboratory 开发^[28,29]。

驱动程序定义了两个网卡操作类 CEcNpfDevice 和 CNpfInfo，通过调用 Winpcap 驱动程序包实现主站通信网卡的管理和 Ethernet 数据帧的收发。

6.2.1 基于 NDIS 的网卡驱动程序

NDIS 是由微软公司和 3Com 公司共同制定的网络接口驱动规范。它在 Windows 操作系统中的位置如图 6.2 所示。

NDIS 为开发网络驱动程序提供了一套标准的接口，使得网络驱动程序的跨平台性更好。NDIS 提供三个层次的接口：

- ① 小端口驱动程序(Miniport driver) 也称为网卡驱动，它向下直接操作网卡硬件，向上为上层驱动提供收发数据帧的接口；
- ② 中间层驱动程序(Intermediate driver, 简称 IMD) 介于协议驱动和小端口驱动之间，可以截获所有的网络数据帧；
- ③ 协议层驱动程序(Protocoldriver) 用于实现一种网络协议栈。

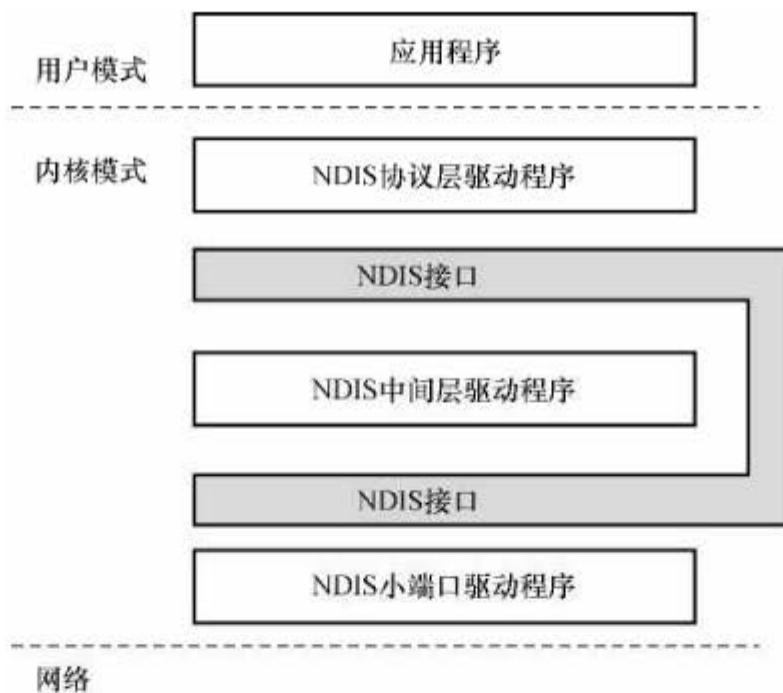


图 6.2 NDIS 结构示意图

WinPcap 为 Win32 应用程序提供访问网络底层的能力，其核心功能是捕获网络数据包。其他功能包括数据包过滤、数据包发送、流量统计和数据包存储等。Winpcap 组成如图 6.3 所示，图中的箭头指向为数据帧的流动方向。

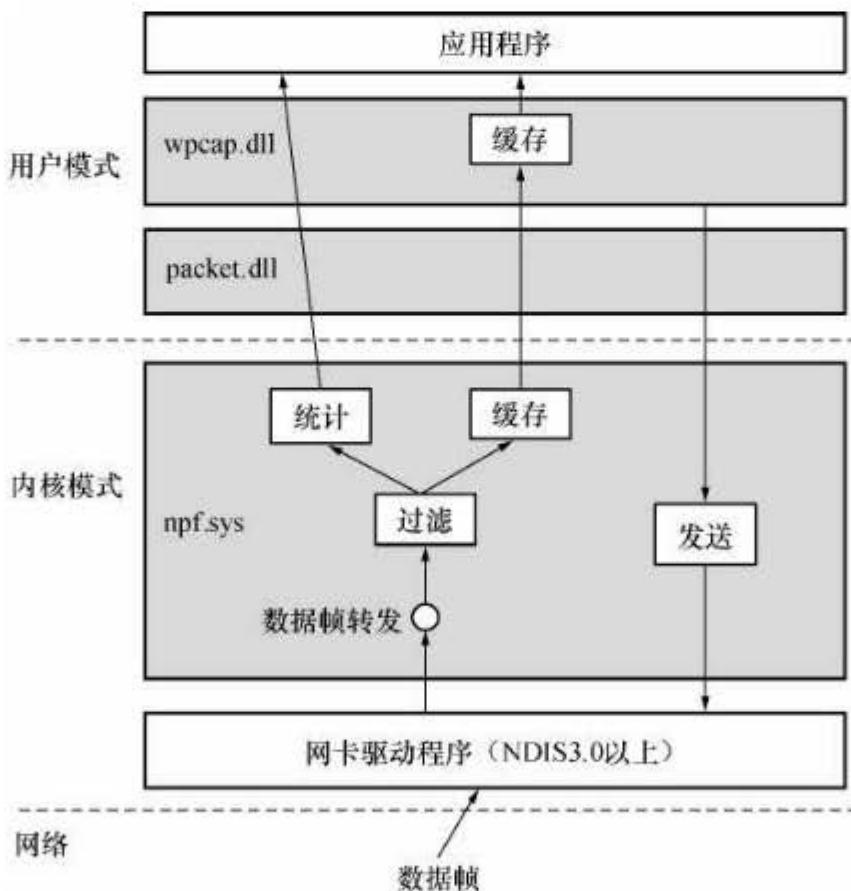


图 6.3 Winpcap 组成

WinPcap 包括三部分内容：

- ① 内核模式的协议驱动程序 npf.sys 其中 npf 为 Netgroup Packet Filter 的简写，实现了高效的网络数据包的捕获和过滤功能，以及网络流量的统计分析，并支持数据包的发送；

② 动态链接库 packet.dll 提供给开发者一个接口，使用它可以调用 WinPcap 的函数，它是一个较底层的开发接口；

③ 动态链接库 wpcap.dll 是一个更高层的编程接口，提供与 Libpcap 兼容的函数调用。

在开发中，为了提高运行效率，EtherCAT 驱动程序调用 packet.dll 直接访问 npf.sys，而不使用 wpcap.dll。packet.dll 有以下 5 种重要函数：

(1) 获得网卡名称

```
internal static extern uint PacketGetAdapterNames(
    IntPtr nameBuffer,           // 网卡名称存储字符串
    int* BufferSize             // 字符串长度数据指针
);
```

(2) 打开网卡

```
internal static extern IntPtr PacketOpenAdapter(
    [MarshalAs(UnmanagedType.LPStr)] string adapterName      // 网卡名称
);
```

(3) 发送一个数据帧

```
internal static extern uint PacketSendPacket(
    IntPtr hAdapterObject,        // 所使用的网卡操作指针
    IntPtr hPacket,               // 要发送的数据帧指针
    uint bSync                   // 是否同步方式
);
```

(4) 接收数据帧

```
internal static extern uint PacketReceivePacket(
    IntPtr pHandle,              // 接收数据包的网络适配器
    IntPtr pPacket,               // 存放收到的数据包
    uint sync                     // 是否同步方式
);
```

(5) 关闭网卡

```
internal static extern uint PacketCloseAdapter(
    IntPtr hAdapterObject        // 要关闭的网络适配器
);
```

6.2.2 CEcNpfDevice 类

CEcNpfDevice 类定义了收发以太网数据帧的相关方法，包括以下函数：

- (1) Open() 打开网卡通信，使用网卡 MAC 地址 m_macAdapter 作为参数；
- (2) SendPacket(PVOID pData, ULONG nData) 发送以太网数据帧，其中，pData 为数据帧地址指针，nData 为数据帧长度；
- (3) StartReceiverThread() 新建一个接收数据帧线程；
- (4) ReceiverThread(LPVOID lpParameter) 接收数据帧线程，接收到的以太网数据帧保存在一个 FiFo 类型的变量 m_listPacket 中；

(5) CheckRecvFrame(PBYTE pData) 从 m_listPacket 中返回一个以太网数据帧到地址 pData;

(6) Close() 关闭所选择的网卡，返回执行结果 virtual HRESULT。

CEcNpfDevice 类定义的程序源代码如下：

```

//-----  

// 网卡操作类CEcNpfDevice类定义  

//-----  

class CEcNpfDevice  

{  

//-----  

// 公有成员函数  

//-----  

public:  

    // 构造函数  

    // 参数 macAdapter: 通信的网卡MAC地址  

    CEcNpfDevice (ETHERNET_ADDRESSmacAdapter =  

        NullEthernetAddress);  

    // 构造函数  

    // 参数 pszAdapter: 通信的网卡名称  

    CEcNpfDevice (LPCSTR pszAdapter);  

    //析构函数  

    virtual ~CEcNpfDevice();  

    // IUnknown  

    virtual ULONG STDMETHODCALLTYPE Release(void);  

    //打开所选择的网卡，返回执行结果  

    virtual HRESULT Open();  

    //关闭所选择的网卡，返回执行结果  

    virtual HRESULT Close();  

    //获得链接波特率  

    virtual ULONG GetLinkSpeed();  

    //向所选用网卡发送一个数据包，并返回调用结果  

    //参数 pData: 要发送数据的指针  

    //参数 nData: 发送数据的字节数

```

```

virtual long SendPacket(PVOID pData, ULONG nData);

//从所接收到的数据帧缓存区得到一个数据帧，并返回调用结果
//参数 pData: 接收的数据保存在pData指针指向的空间中
virtual long CheckRecvFrame(PBYTE pData);

//-----  

// 保护的成员函数  

//-----  

protected:  

    //从所选网卡读取数据帧，保存在fifo列表m_listPacket中
    virtual long ReadPackets();

//-----  

// 保护的成员变量  

//-----  

LPSTR          m_pszAdapter; // 选用的网卡名称
ETHERNET_ADDRESS m_macAdapter; // 选用的网卡MAC地址

//-----  

// 私有成员函数  

//-----  

private:  

    //创建一个线程，从网卡接收以太网数据帧，并返回调用结果
    //参数：nPriority: 线程的优先级
    long StartReceiverThread(long nPriority =
        THREAD_PRIORITY_HIGHEST);

    //线程函数，返回调用结果
    //参数：lpParameter: 线程参数
    static DWORD WINAPI ReceiverThread(LPVOID lpParameter);

//-----  

// 私有成员变量  

//-----  

HANDLE          m_hStartEvent;
HANDLE          m_hCloseEvent;
HANDLE          m_hReceiverThread; // 接收线程句柄
DWORD           m_dwThreadId;
bool            m_bStopReceiver; // 运行标志

```

```

long          m_lRef;
struct _ADAPTER*   m_pAdapter;      // 选中的网卡的操作信息
CFiFoList<PVOID, MAX_NPFPACKETS>m_listPacket;
                                         //接收缓存列表，先入先出
};

```

6.2.3 CNpfInfo类

CNpfInfo类定义了获得网卡信息的相关方法，包括以下函数：

- (1) GetAdapterCount() 获得当前计算机网卡数目；
- (2) GetAdapterName(int nAdapter) 根据网卡编号nAdapter获得网卡名称；
- (3) GetAdapterDescription(int nAdapter) 根据网卡编号nAdapter获得网卡描述信息；
- (4) GetAdapterInfo() 获得计算机所有网卡的信息，并保存到数组变量m_pAdapterInfo[MAX_NUM_ADAPTER]；

CNpfInfo类定义的程序源代码如下：

```

//-----  

// 网卡信息类CNpfInfo定义  

//-----  

class CNpfInfo  

{  

//-----  

// 公有成员函数  

//-----  

public:  

    CNpfInfo();      // 构造函数  

    ~CNpfInfo();    // 析构函数  

    int GetAdapterCount(); // 获得当前计算机网卡数目  

    //获得网卡名称  

    //参数： nAdapter: 网卡编号  

    LPCSTR GetAdapterName(int nAdapter);  

    //获得网卡描述  

    //参数： nAdapter: 网卡编号  

    LPCSTR GetAdapterDescription(int nAdapter);  

    //获得计算机所有网卡的信息  

    BOOL GetAdapterInfo();  

    //根据网卡MAC地址打开网卡并得到操作信息
}

```

```

struct _ADAPTER* GetAdapter(ETHERNET_ADDRESS macAddress);

//根据网卡名称打开网卡并得到操作信息
struct _ADAPTER* GetAdapter(LPCSTR pszAdapter,
    ETHERNET_ADDRESS &macAddress);

//根据网卡编号打开网卡并得到操作信息
struct _ADAPTER* GetAdapter(int nAdapter,
    ETHERNET_ADDRESS &macAddress);

//-----  

// 保护成员变量  

//-----  

protected:  

// 网卡信息
    EcAdapterInfo      m_pAdapterInfo[MAX_NUM_ADAPTER];
    int                  m_nAdapter;           // 网卡数目
};
```

6.2.4 获得计算机网卡信息

CNpfInfo::GetAdapterInfo 函数调用 packet.dll 中的 PacketGetAdapterNames() 函数以获得本机所有网卡的名称，并最终保存在结构体数组 m_pAdapterInfo 中。其程序的源代码如下：

```

//-----  

// CNpfInfo::GetAdapterInfo(): 获得计算机上所有网卡的信息  

// m_nAdapter: 存放网卡数目  

// m_pAdapterInfo[MAX_NUM_ADAPTER]: 存放网卡信息  

//-----  

BOOL CNpfInfo::GetAdapterInfo()  

{
    int i=0;
    char *szTmpName,*szTmpName1;
    ULONG nAdapterLength = DEFAULT_ADAPTER_NAMELIST;
    char *szAdapterName = new char[nAdapterLength];
    m_nAdapter = -1;
    // 获得网卡名称
    if(PacketGetAdapterNames(PTSTR(szAdapterName),
    &nAdapterLength)==FALSE)
    {   // 如果函数执行失败, 清空szAdapterName地址空间
        delete[] szAdapterName;
        szAdapterName = new char[nAdapterLength];
```

```

    if (PacketGetAdapterNames (PTSTR (szAdapterName) ,
&nAdapterLength)==FALSE)
    {   // 函数再次执行失败，返回错误结果
        delete [] szAdapterName;
        return FALSE;
    }
}

szTmpName=szAdapterName;
szTmpName1=szAdapterName;

// 顺序得到本机网卡名称
while (((*szTmpName]!='\0') || (* (szTmpName-1) !='\0')))
{
    if (*szTmpName=='\0')
    {
        if( m_pAdapterInfo[i].szName )
            delete[] m_pAdapterInfo[i].szName;
        m_pAdapterInfo[i].szName = new
            char [szTmpName-szTmpName1+1];
        strcpy( m_pAdapterInfo[i].szName, szTmpName1 );
        szTmpName1=szTmpName+1;
        i++;
    }
    szTmpName++;
}
m_nAdapter=i;
szTmpName++;

// 顺序得到本机网卡描述
for( i=0; i<m_nAdapter; i++)
{
    if( m_pAdapterInfo[i].szName )
        delete[] m_pAdapterInfo[i].szDescr;
    m_pAdapterInfo[i].szDescr = new
        char [strlen(szTmpName)+1];
    strcpy(m_pAdapterInfo[i].szDescr, szTmpName);
    szTmpName += (strlen (szTmpName)+1);
}
delete [] szAdapterName;

```

```

    return TRUE;
}

```

6.2.5 打开网卡

CEcNpfDevice::Open()函数根据给定的MAC地址查找相应的网卡，并得到其操作指针；然后需要逐个打开本地网卡，得到其网卡MAC地址，并与给定地址相比较；直到地址匹配，得到其LPADAPTER类型的地址指针m_pAdapter，在发送和接收数据帧时都使用此指针操作相应网卡。CEcNpfDevice::Open()函数的调用如图6.4所示。

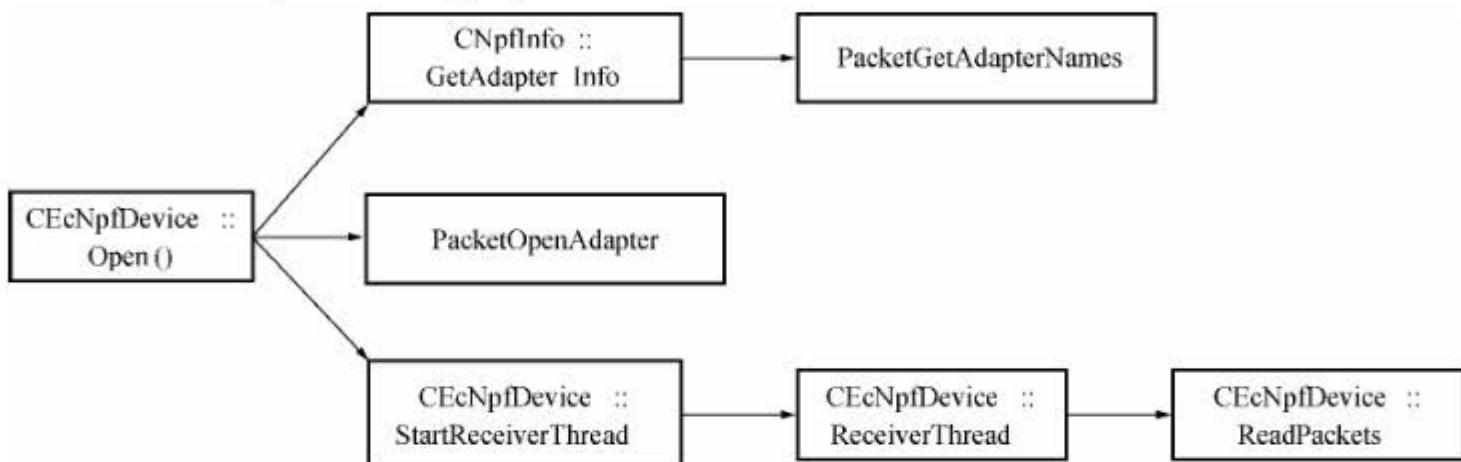


图 6.4 接收数据帧流程调用图

CEcNpfDevice::Open()函数的程序源代码如下：

```

//-----  

// CEcNpfDevice::Open(): 根据网卡MAC地址打开网卡并得到操作信息  

// m_macAddress: 要操作的网卡MAC地址，在新建CEcNpfDevice数据对象时获得  

// LPADAPTER m_pAdapter: 得到网卡操作指针  

//-----  

HRESULT CEcNpfDevice::Open()  

{
    HRESULT hr = EC_E_ERROR;  

    if( m_pAdapter )  

        return EC_E_INVALIDSTATE;  

    CNpfInfo npfInfo;           // 定义一个CNpfInfo类数据对象  

    LPADAPTER pAdapter = NULL; // 定义指向网卡的指针  

    PPACKET_OID_DATA pOidData; // 网卡OID数据对象  

    npfInfo.GetAdapterInfo();   // 获得当前计算机网卡信息，见6.2.4节  

    m_nAdapter = npfInfo.m_nAdapter;// 得到当前计算机网卡数目  

    // 为pOidData分配内存空间  

    pOidData = (PPACKET_OID_DATA) new  

    BYTE[sizeof(ETHERNET_ADDRESS)+sizeof(PACKET_OID_DATA)];  

    if (pOidData == NULL)  

        return NULL;
}

```

```

// 顺序打开本机上的网卡，查找MAC匹配的网卡，得到其操作指针，并保存到
// 变量m_pAdapter；不匹配网卡则被关闭
for( int i=0; i<m_nAdapter; i++ )
{
    pAdapter = PacketOpenAdapter( (char*)GetAdapterName(i));
    if ( !pAdapter || (pAdapter->hFile ==
INVALID_HANDLE_VALUE) )
    {
        // 如果打开网卡不成功
        DWORD dwErrorCode=GetLastError();
        TRACE(_T("Unable to open the adapter,
                Error Code : %lx\n"),dwErrorCode);
        pAdapter = NULL;
        continue;
    }

    // 初始化pOidData结构体
    pOidData->Oid = OID_802_3_CURRENT_ADDRESS;
    pOidData->Length = sizeof(ETHERNET_ADDRESS);
    ZeroMemory(pOidData->Data, sizeof(ETHERNET_ADDRESS));
    // 比较当前打开网卡MAC地址和给定MAC地址
    if( PacketRequest(pAdapter, FALSE, pOidData) &&
        memcmp(&m_macAddress, pOidData->Data,
               sizeof(ETHERNET_ADDRESS)) == 0 )
    {
        // 如果MAC地址匹配，则停止查找，跳出循环
        break;
    }
    PacketCloseAdapter(pAdapter);
    pAdapter = NULL;
}

m_pAdapter = pAdapter;

// 如果获得有效网卡指针，则启动数据帧接收线程
if( m_pAdapter )
{
    StartReceiverThread();      // 启动接收数据帧线程
    ::WaitForSingleObject(m_hStartEvent, INFINITE);
    hr = S_OK;
}
return hr;

```

}

在打开网卡的同时，启动一个线程以接收数据帧，启动线程函数的源代码如下：

```

//-----  

// CEcNpfDevice::StartReceiverThread(): 启动接收数据帧线程  

// nPriority: 新建线程的优先级  

//-----  

long CEcNpfDevice::StartReceiverThread(long nPriority)  

{  

    long nErr=ERROR_SUCCESS;  

    m_bStopReceiver = false;  

    // 创建一个新线程，线程函数为ReceiverThread()  

    m_hReceiverThread = ::CreateThread(NULL, 0, ReceiverThread,  

                                         this, CREATE_SUSPENDED, &m_dwThreadId);  

    ASSERT(m_hReceiverThread);  

    if (m_hReceiverThread)  

    {  

        BOOL bResult;  

        // 设置线程的优先级  

        bResult=::SetThreadPriority(m_hReceiverThread, nPriority);  

        ASSERT(bResult);  

        // 启动线程  

        ::ResumeThread(m_hReceiverThread);  

    }  

    else  

        nErr = ::GetLastError();  

    return nErr;  

}

```

接收数据帧的线程函数ReceiverThread()接收和保存网卡上收到的以太网数据帧，它由ReadPackets()函数实现，将在6.2.7节中对它做详细介绍。ReceiverThread()函数的源代码如下：

```

//-----  

// CEcNpfDevice::ReceiverThread(): 接收数据帧线程的实现函数  

//-----  

DWORD WINAPI CEcNpfDevice::ReceiverThread(LPVOID lpParameter)  

{  

    ASSERT(lpParameter);  

    if (lpParameter == NULL)  

        return -1;
}

```

```

CEcNpfDevice* pThis = (CEcNpfDevice*)lpParameter;
return pThis->ReadPackets(); // 见6.2.7节
}

```

6.2.6 发送数据帧

CEcNpfDevice::SendPacket()函数用以发送以太网数据帧，其主要步骤如下：

- (1) 调用函数PacketAllocatePacket(), 用以创建一个数据帧结构pPacket;
- (2) 调用函数PacketInitPacket(), 用发送数据指针pData和发送数据字节数nData初始化数据帧结构pPacket;
- (3) 调用PacketSendPacket(), 用以发送数据帧;
- (4) 调用PacketFreePacket(pPacket), 用以释放数据帧所使用内存。

CEcNpfDevice::SendPacket()函数的程序源代码如下：

```

// -----
// CEcNpfDevice:: SendPacket (): 发送数据帧
// pData: 要发送的数据存在应用程序地址pData
// nData: 要发送的数据帧字节数
// -----
long CEcNpfDevice::SendPacket (PVOID pData, ULONG nData)
{
    if( m_pAdapter == NULL )
        return ECERR_DEVICE_INVALIDSTATE;

    LPPACKET pPacket;

    // 创建一个数据帧结构pPacket
    if((pPacket = PacketAllocatePacket())==NULL)
    { // 函数执行失败, 返回错误代码ECERR_DEVICE_NOMEMORY
        TRACE ("TCEcNpfDevice::SendPacket: Error:
                failed to allocate the LPPACKET structure.\n");
        return ECERR_DEVICE_NOMEMORY;
    }

    // 用pData和nData初始化数据帧结构pPacket
    PacketInitPacket (pPacket, pData, nData);

    // 向网卡m_pAdapter发送数据结构pPacket
    if(PacketSendPacket (m_pAdapter,pPacket,TRUE)==FALSE)
    { // 函数执行失败, 返回错误代码ECERR_DEVICE_SENDFAILED

```

```

TRACE( "TCEcNpfDevice::SendPacket: Error:
        failed to send packet.\n");
// 释放数据帧结构pPacket
PacketFreePacket(pPacket);
return ECERR_DEVICE_SENDFAILED;
}
PacketFreePacket(pPacket);

return ECERR_NOERR;
}

```

6.2.7 接收数据帧

数据帧的接收使用了多线程编程技术，CEcNpfDevice类在打开网口的同时启动了一个接收以太网数据帧的线程，这个线程由CEcNpfDevice::ReadPackets()函数实现。

接收线程首先调用Winpcap函数将网卡配置为混杂（PROMISCUOUS）模式，以便接收所有以太网数据帧；然后设置Winpcap以接收缓存区等参数，最后等待接收以太网数据帧，并将数据帧保存在FiFo类型的变量m_listPacket中。接收数据帧线程的源代码如下：

```

//-----//
// CEcNpfDevice::ReadPackets(): 从所选择网卡读取数据帧，保存在fifo列
//                                表m_listPacket中
// m_pAdapter: 要操作的网卡的操作信息
// LPADAPTER pAdapter: 返回网卡操作信息
//-----//
long CEcNpfDevice::ReadPackets()
{
    LPPACKET pPacket;
    char buffer[256000];
    HANDLE hEvents[2];

    if( m_pAdapter == NULL ) // 如果没有打开有效的网卡
        return ECERR_DEVICE_INVALIDSTATE;

    // 设置网卡读取所有的本地数据包，包括广播帧和寻址到本MAC地址的帧
    if( PacketSetHwFilter(m_pAdapter,
        NDIS_PACKET_TYPE_PROMISCUOUS)==FALSE )
    { // 函数执行失败，不能设置为混杂模式
        TRACE(_T("CEcNpfDevice::ReadPackets:
                  unable to set promiscuous mode!\n"));
    }
}

```

```

// 设置内核缓冲区中激发本事件的最小数据长度
if( PacketSetMinToCopy(m_pAdapter,1)==FALSE )
{
    // 函数执行失败，返回错误码ECERR_DEVICE_ERROR
    TRACE(_T("CEcNpfDevice::ReadPackets:
              Unable to set min copy!\n"));
    return ECERR_DEVICE_ERROR;
}

// 设置缓存区为512KB
if( PacketSetBuff(m_pAdapter,512000)==FALSE )
{
    // 函数执行失败，返回错误码ECERR_DEVICE_ERROR
    TRACE(_T("CEcNpfDevice::ReadPackets:
              Unable to set the kernel buffer!\n"));
    return ECERR_DEVICE_ERROR;
}

// 设置读操作的超时时间
if( PacketSetReadTimeout(m_pAdapter, INFINITE)==FALSE)
{
    // 函数执行失败，不能设置超时时间
    TRACE(_T("CEcNpfDevice::ReadPackets:Warning:
              unable to set the read tiemout!\n"));
}

// 分配并初始化数据区结构用来接收数据
if( (pPacket = PacketAllocatePacket())==NULL )
{
    // 函数执行失败，返回错误码ECERR_DEVICE_NOMEMORY
    TRACE(_T("CEcNpfDevice::ReadPackets Error:
              failed to allocate the LPPACKET structure.\n"));
    return ECERR_DEVICE_NOMEMORY;
}
PacketInitPacket(pPacket, (char*)buffer, sizeof(buffer));

::SetEvent(m_hStartEvent);
//set events to wait for
hEvents[0] = m_hCloseEvent;
hEvents[1] = PacketGetReadEvent(m_pAdapter);

while( !m_bStopReceiver ) // 开始接收数据包
{

```

```

switch( ::WaitForMultipleObjects(2, hEvents, FALSE,
INFINITE) ) // 等待事件发生
{
// 关闭网卡事件
case WAIT_OBJECT_0:
break;
// 读数据包事件
case WAIT_OBJECT_0 + 1:
{
// 从网卡m_pAdapter读数据帧到数据区pPacket
if( !PacketReceivePacket(m_pAdapter, pPacket, TRUE) )
break; // 接收数据包失败, 终止当前循环

if( m_bStopReceiver )
break; // 如果停止接收, 终止当前循环

ULONG nBytesReceived;
BYTE *pBuf; // 数据区指针
ULONG nOff=0;
struct bpf_hdr *pHdr;

// 得到接收数据区的大小
nBytesReceived = pPacket->ulBytesReceived;
pBuf = (BYTE*)pPacket->Buffer;

nOff=0;
// 从头开始处理接收到数据包
while( nOff<nBytesReceived)
{
pHdr=(struct bpf_hdr *) (pBuf+nOff);
ASSERT(pHdr->bh_datalen == pHdr->bh_caplen);
// 新建一个数据帧pNewHdr
struct bpf_hdr *pNewHdr = (struct bpf_hdr *)new
BYTE[sizeof(bpf_hdr)+pHdr->bh_caplen];
// 读取数据帧头
memcpy(pNewHdr, pHdr, sizeof(bpf_hdr));
// 读取数据帧数据
memcpy((BYTE*)pNewHdr+sizeof(bpf_hdr),
(BYTE*)pHdr+pHdr->bh_hdrlen, pHdr->bh_caplen);
pNewHdr->bh_hdrlen=sizeof(bpf_hdr);
}
}

```

```

    // 将数据帧添加到列表m_listPacket
    if( ! m_listPacket.Add(pNewHdr) )
    {
        // 添加失败，删除当前指针
        delete[] pNewHdr;
    }
    // 指针向后移动
    nOff=Packet_WORDALIGN(nOff+pHdr->bh_hdrlen
        + pHdr->bh_caplen);
}
break;
}
}
return 0;
}

```

在应用程序中，调用函数 CEcNpfDevice::CheckRecvFrame(PBYTE pData)，从 m_listPacket 得到一个数据帧，并保存到指针 pData 所指的空间，其程序源代码如下：

```

//-----
// CEcNpfDevice::CheckRecvFrame (): 应用程序获得一个数据帧
// pData: 数据帧存在应用程序地址pData
//-----
long CEcNpfDevice::CheckRecvFrame(PBYTE pData)
{
    int nData;
    PVOID pTemp;

    if(m_listPacket.Remove(pTemp))
    {
        struct bpf_hdr* pHdr = (struct bpf_hdr*) pTemp;
        nData = pHdr->bh_caplen;
        // 将数据帧复制到应用程序地址pData
        memcpy(pData, ((BYTE*) pHdr + sizeof(bpf_hdr)), nData);
        delete pTemp;
        return nData;
    }
    return 0;
}

```

6.2.8 关闭网卡

在退出主站程序之前，需要关闭已经打开的网卡，关闭网卡之前需要终止接收线程，关闭之后清空接收列表变量 m_listPacket，以释放内存，其程序源代码如下：

```

// -----
// CEcNpfDevice::Close ()：关闭网卡
// -----
HRESULT CEcNpfDevice::Close()
{
    if( m_hReceiverThread )
    {
        m_bStopReceiver = true;
        // 通知接收线程自动终止
        ::SetEvent(m_hCloseEvent);
        DWORD dwWaitResult;
        // 等待直到线程终止
        switch ( dwWaitResult = ::WaitForSingleObject
            (m_hReceiverThread, INFINITE) )
        {
            case WAIT_FAILED:
                break;
            case WAIT_OBJECT_0:
                TRACE(_T("CEcNpfDevice::Close; Wait for thread
                    (0x%x) exit successful !\n"), m_dwThreadId);
                break;
            case WAIT_TIMEOUT:
                TRACE(_T("CEcNpfDevice::Close;
                    Timeout ::WaitForSingleObject elapsed !\n"));
                break;
            // 终止接收线程
            ::TerminateThread(m_hReceiverThread, ~0);
            break;
        default:
            TRACE(_T("CEcNpfDevice::Close; Unknown error
                occurred by WaitForSingleObject!\n"));
            ::TerminateThread(m_hReceiverThread, ~0);
            break;
        }
        ::CloseHandle(m_hReceiverThread);
    }
}

```

```

    m_hReceiverThread = NULL;
    ::ResetEvent(m_hCloseEvent);
}

if( m_pAdapter )
{
    // 关闭网卡m_pAdapter
    PacketCloseAdapter(m_pAdapter);

}

PVOID pData;
// 清除数据帧列表m_listPacket
while( m_listPacket.Remove(pData) )
    delete pData;

return S_OK;
}

```

关闭网卡函数在 `CEcNpfDevice` 类的析构函数中调用，而析构函数则是在程序关闭退出之前由系统自动调用。`CEcNpfDevice` 类析构函数的源代码如下：

```

CEcNpfDevice::~CEcNpfDevice()
{
    Close();                                // 关闭网卡通信
    CloseHandle(m_hCloseEvent);
    safe_delete_a(m_pszAdapter);
    ULONG cbRead;
    while( ReadPacket(NULL, 0, cbRead) == ECERR_NOERR )
    {
        // 等待直到线程停止
    }
}

```

6.3 从站设备对象的定义和实现

`CEcSimSlave` 类实现了从站基本通信配置数据的定义和初始化，本驱动程序示例定义了两种类型的从站：

- ① 微处理器接口的 EtherCAT 从站 使用 4 个 SM 通道，支持邮箱通信和周期性数据通信；
- ② 直接 I/O 控制 EtherCAT 从站 使用 16 位输入和 16 位输出。

6.3.1 CEcSimSlave 类的定义

CEcSimSlave 类主要定义了从站配置数据，以及其他状态数据和控制数据，成员函数只有构造函数和析构函数，其程序源代码如下：

(1) 同步管理器通道 SM 结构体

```
//-----  
// TSYNCMANAGE: SYNCMANAGE数据结构定义  
//-----  
  
typedef struct TSYNCMANAGE  
{  
    USHORT m_nPhyStart;           // 物理起始地址  
    USHORT m_nLength;            // 数据长度  
    UCHAR m_cControl;           // 控制寄存器  
    UCHAR m_cStatus;             // 状态寄存器  
    UCHAR m_cActive;              // ECAT端激活寄存器  
    UCHAR m_cPdiControl;         // PDI端控制寄存器  
} SYNCMANAGE, *PSYNCMANAGE;
```

(2) FMMU 配置结构体

```
//-----  
// TFMMU: FMMU数据结构定义  
//-----  
  
typedef struct TFMMU  
{  
    UINT   m_nLgStart; // 逻辑起始地址  
    USHORT m_nLength; // 数据长度  
    UCHAR  m_cLgStartBit; // 逻辑起始位  
    UCHAR m_cLgStopBit; // 逻辑终止位  
    USHORT m_nPhyStart; // 物理内存起始地址  
    UCHAR m_cPhyStartBit; // 物理起始位  
    UCHAR m_cType; // 类型  
    UCHAR m_cActive; // 激活  
    USHORT m_nReserve; // 保留数据  
} FMMU, *PFMMU;
```

(3) CEcSimSlave类定义

```
//-----  
// CEcSimSlave: 从站类定义，定义了主站需要配置从站的信息  
//-----  
  
class CEcSimSlave  
{
```

```

//-----  

// 公有成员函数  

//-----  

public:  

    CEcSimSlave(int slvType); // 构造函数  

    ~CEcSimSlave(void); // 析构函数  

//-----  

// 公有成员变量  

//-----  

public:  

    FMMU          m_pFmmu[4];      // FMMU  

    SYNCMANAGE   m_pSyncM[4];     // SYNCMANAGER  

    USHORT        m_cStatus;       // 状态字  

    USHORT        m_nStatusCode;   // 状态码  

    USHORT        m_nDlAddr;        // 数据链路层地址  

    int           m_nSlvType;      // 从站类型  

    USHORT        m_nOutOffset;    // 从站输出数据存储偏移地址  

    USHORT        m_nOutLen;       // 从站输出数据长度  

    USHORT        m_nInOffset;     // 从站输入数据存储偏移地址  

    USHORT        m_nInLen;        // 从站输入数据长度  

};

```

6.3.2 CEcSimSlave 类的实现

CEcSimSlave 类只实现了其构造函数，分别对两种类型的 EtherCAT 从站做配置数据初始化。两种类型的从站 SM 默认配置如表 6.1 所列，每个从站设备对象在定义的时候就按照表中的默认值进行初始化，应用程序可以修改微处理器接口的从站周期性数据通信 SM 通道配置。

表 6.1 SM 通道基本配置

SM 通道	微处理器接口的从站	直接 16 位输入和 16 位输出从站
0	邮箱输出， 物理起始地址：0x1800 数据长度：64 字节 1 个缓存区，写操作	数字量输出数据， 物理起始地址：0x0f00 数据长度：2 字节 1 个缓存区，写操作
1	邮箱输入， 物理起始地址：0x1C00 数据长度：64 字节 1 个缓存区，读操作	数字量输入数据， 物理起始地址：0x1000 数据长度：2 字节 3 个缓存区，读操作

续表 6.1

SM 通道	微处理器接口的从站	直接 16 位输入和 16 位输出从站
2	过程数据输出, 物理起始地址: 0x1000 数据长度: 16 字节 3 个缓存区, 写操作	不使用
3	过程数据输入, 物理起始地址: 0x1100 数据长度: 16 字节 3 个缓存区, 读操作	不使用

CEcSimSlave 类构造函数的程序源代码如下:

```

//-----  

// CEcSimSlave类的构造函数  

// slvType: 从站类型  

// 1: 8位并行微处理器总线接口  

// 0: 16IN/16OUT数字量I/O从站  

//-----  

CEcSimSlave::CEcSimSlave(int slvType)  

{  

    int i;  

    m_cStatus= 1;  

    m_nStatusCode = 0;  

    m_nDlAddr = 1000;  

    m_nSlvType = slvType;  

    m_nOutOffset = 0;  

    m_lOut = 0;  

    m_nOutLen = 0;  

    m_nInOffset = 0;  

    m_lIn = 0;  

    m_nInLen = 0;  

    for (i=0;i<4;i++)  

    {  

        m_pSyncM[i].m_cStatus = 0;  

        m_pSyncM[i].m_cActive = 0x01;  

        m_pSyncM[i].m_cPdiControl = 0;  

    }  

    if (slvType == ISASLAVE) // 8位并行微处理器总线接口
}

```

```

    {
        m_pSyncM[0].m_nPhyStart = 0x1800;
        m_pSyncM[0].m_nLength = 64;
        m_pSyncM[0].m_cControl = 0x26;

        m_pSyncM[1].m_nPhyStart = 0x1C00;
        m_pSyncM[1].m_nLength = 32;
        m_pSyncM[1].m_cControl = 0x22;

        m_pSyncM[2].m_nPhyStart = 0x1000;
        m_pSyncM[2].m_nLength = 64;
        m_pSyncM[2].m_cControl = 0x24;

        m_pSyncM[3].m_nPhyStart = 0x1100;
        m_pSyncM[3].m_nLength = 16;
        m_pSyncM[3].m_cControl = 0x20;
    }

    else //I/O从站接口
    {
        m_pSyncM[0].m_nPhyStart = 0x0f00;
        m_pSyncM[0].m_nLength = 2;
        m_pSyncM[0].m_cControl = 0x46;
        m_pSyncM[0].m_cActive = 0x0;

        m_pSyncM[1].m_nPhyStart = 0x1000;
        m_pSyncM[1].m_nLength = 2;
        m_pSyncM[1].m_cControl = 0x0;

        m_pSyncM[2].m_nPhyStart = 0;
        m_pSyncM[2].m_nLength = 0;
        m_pSyncM[2].m_cControl = 0x24;

        m_pSyncM[3].m_nPhyStart = 0;
        m_pSyncM[3].m_nLength = 0;
        m_pSyncM[3].m_cControl = 0x20;
    }
}

```

6.4 主站设备对象的定义和实现

6.4.1 CEcSimMaster 类的定义

CEcSimMaster 类是驱动程序的主体类，它定义了 CEcNpfDevice 类型的指针变量 m_pNpfdev 以实现网卡控制；定义了 CEcSimSlave 类型的指针变量 m_ppEcSlave 来管理从站；周期性输入和输出数据分别用字节型数组 m_InputImage 和 m_OutputImage 存放。CEcSimMaster 类中的重要成员函数有：

- ① Open() 启动主站运行，打开网卡通信；
- ② CreatSlave() 新建一个从站数据对象，并写入配置数据；
- ③ ImageAssign() 根据各个从站的 SM 配置情况自动配置从站 FMMU 参数；
- ④ StateMachine() 处理 EtherCAT 状态机；
- ⑤ PrepareCyclicFrameFmmu() 准备逻辑寻址周期性发送数据帧框架；
- ⑥ PrepareCyclicFrame() 准备设置寻址周期性发送数据帧框架；
- ⑦ SendCyclicFrameFmmu() 使用逻辑寻址发送周期性数据命令，各从站使用 FMMU 映射输入输出数据；
- ⑧ SendCyclicFrame() 发送周期性数据命令；
- ⑨ CheckFrames() 从 CEcNpfDevice 类接收并处理返回数据帧。

CEcSimMaster类定义的程序源代码如下：

```

//-----  

// CEcSimMaster类定义  

//-----  

class CEcSimMaster  

{  

//-----  

// 公共成员函数  

//-----  

public:  

    CEcSimMaster(int n);      // 构造函数  

    ~CEcSimMaster(void);     // 析构函数  

    void CreatDevice();       // 构造CEcNpfDevice类  

    // 根据输入参数新建一个从站类对象  

    void CreatSlave(int i, int nType, USHORT nSt1, USHORT nLen1,  

                    USHORT nSt2, USHORT nLen2);  

    bool Open();                // 打开网卡设备  

    void Delay(int i);          // 延时函数  

    void WriteSM(int i);        // 写SM配置数据，参数i为SM编号  

    void ClearSyncM(BYTE state); // 根据当前状态清除SM配置数据  

    void ActiveOutput(BOOL bOut); // 激活从站输出
}

```

```

// 写Fmmu配置数据，参数i为Fmmu编号
void WriteFmmu(int i);

// 将请求状态state写入AL控制寄存器
void WriteAlControl(BYTE state);

// 读从站当前状态
void ReadAlState();           // 读从站当前状态
void WriteDlAddr();           // 为从站配置地址
void StateMachine();          // 执行状态机处理
void Release();                // 释放网卡设备
void PrepareCyclicFrameFmmu() // 准备逻辑寻址周期性发送数据帧框架
void PrepareCyclicFrame()     // 准备设置寻址周期性发送数据帧框架
void SendCyclicFrame();       // 发送周期性数据命令
void SendCyclicFrameFmmu();   // 发送周期性数据命令，使用FMMU
long CheckFrames();           // 接收并检查返回数据帧
void ImageAssign();           // FMMU配置
int GetStateMachine();         // 得到当前状态
CString GetStateString(USHORT nState); // 得到当前状态字符串

//-----公共成员变量-----public:
int      m_nStatus;           // 当前状态
int      m_nRequire;          // 请求状态
USHORT   m_nReadStatus;        // 读到从站实际状态
long     m_lSendFrame;         // 发送数据帧计数
long     m_lRecvFrame;         // 接收数据帧计数
BOOL     m_bFmmu;              // 是否使用FMMU标志
int      m_nEth;               // 使用网卡编号
ULONG    m_nEcSlave;           // 从站数目
USHORT   m_nCycTime;           // 周期时间
BYTE     m_nIndex;              // EtherCAT命令索引
BYTE     m_InputImage[MAXIMAGESIZE]; // 输入数据映射区
BYTE     m_OutputImage[MAXIMAGESIZE]; // 输出数据映射区
USHORT   m_nInSize;             // 输入数据字节数
USHORT   m_nOutSize;            // 输出数据字节数
CEcSimSlave** m_ppEcSlave;    // 从站对象指针
CEcNpfDevice * m_pNpfdev;      // 网卡设备指针
ETHERNET_ADDRESS m_macAddr;    // MAC地址
};


```

6.4.2 初始化和启动 CEcSimMaster 数据对象

在定义一个 CEcSimMaster 数据对象的时候运行 CEcSimMaster 类构造函数，完成其成员变量数据的初始化，该初始化的程序源代码如下：

```

//-----  

// CEcSimMaster类定义  

// 参数n: 从站数目  

//-----  

CEcSimMaster::CEcSimMaster(int n)  

{  

    int i;  

    m_nEcSlave = n;  

    m_nStatus = 10;  

    m_nRequire = 10;  

    m_nIndex = 0;  

    m_lSendFrame = 0;  

    m_lRecvFrame = 0;  

    m_bFmmu = 1;  

    m_nEth = 0;  

    m_ppEcSlave           = new CEcSimSlave*[m_nEcSlave];  

    memset(m_ppEcSlave, 0, m_nEcSlave*sizeof(CEcSimSlave*));  

    for (i=0;i<m_nEcSlave;i++)  

    {  

        m_ppEcSlave[i] = new CEcSimSlave(IOSLAVE);  

        m_ppEcSlave[i]->m_nDlAddr = 1000 + i;  

    }  

    memset(m_OutputImage, 0, MAXIMAGESIZE);  

    memset(m_InputImage, 0, MAXIMAGESIZE);  

    m_pNpfdev = NULL;  

}

```

主站类中使用 Open() 函数新建一个网口通信控制数据对象，并打开网卡通信，启动主站运行。Open() 函数的程序源代码如下：

```

bool CEcSimMaster::Open()  

{  

    HRESULT hr;  

    // 使用给定MAC地址新建一个网口通信控制数据对象  

    m_pNpfdev = new CEcNpfDevice(m_macAddr);  

    if (m_pNpfdev)  

    {

```

```

//打开网卡通信
if( !SUCCEEDED(hr = m_pNpfdev->Open()) )
{
    return FALSE;
}
return TRUE;
}

```

6.4.3 配置从站设备对象

主站设备类根据系统配置数据配置从站设备对象，只要正确配置从站ESC的SM通道参数即可实现基本的数据通信，如果使用逻辑寻址的话，需要根据各个从站的过程数据来设定逻辑寻址命令和各从站的FMMU通道参数。该配置过程的程序源代码如下。

(1) SM通道配置

```

//-----
//从站类对象参数刷新
//i:          从站编号
//nType:      从站类型
//nSt1:       输出数据SM起始地址
//nLen1:      输出数据SM长度
//nSt2:       输入数据SM起始地址
//nLen2:      输入数据SM长度
//-----

void CEcSimMaster::CreatSlave(int i, int nType, USHORT nSt1,
    USHORT nLen1, USHORT nSt2, USHORT nLen2)
{
    m_ppEcSlave[i] = new CEcSimSlave(nType);
    m_ppEcSlave[i]->m_pSyncM[2].m_nPhyStart = nSt1;
    m_ppEcSlave[i]->m_pSyncM[2].m_nLength = nLen1;
    m_ppEcSlave[i]->m_pSyncM[3].m_nPhyStart = nSt2;
    m_ppEcSlave[i]->m_pSyncM[3].m_nLength = nLen2;
    m_ppEcSlave[i]->m_nD1Addr = 1000 + i;
}

```

(2) FMMU通道配置

```

//-----
//计算过程数据使用FMMU配置
//-----

void CEcSimMaster::ImageAssign()
{

```

```

int i;

//-----  

// 从站0输出FMMU配置  

//-----  

// 输出FMMU通道长度等于输出SM通道长度  

m_ppEcSlave[0]->m_nOutLen = m_ppEcSlave[0]->  

    m_pSyncM[2*m_ppEcSlave[0]->m_nSlvType].m_nLength;  

// 输入FMMU通道长度等于输入SM通道长度  

m_ppEcSlave[0]->m_nInLen = m_ppEcSlave[0]->  

    m_pSyncM[2*m_ppEcSlave[0]->m_nSlvType + 1].m_nLength;  

// 输出FMMU通道逻辑起始地址为0x00001000  

m_ppEcSlave[0]->m_pFmmu[0].m_nLgStart = 0x00001000;  

// 输出FMMU通道数据长度  

m_ppEcSlave[0]->m_pFmmu[0].m_nLength =  

    m_ppEcSlave[0]->m_nOutLen;  

// 输出FMMU通道逻辑起始位等于0  

m_ppEcSlave[0]->m_pFmmu[0].m_cLgStartBit = 0;  

// 输出FMMU通道逻辑终止位等于7  

m_ppEcSlave[0]->m_pFmmu[0].m_cLgStopBit = 7;  

// 输出FMMU通道物理起始地址等于输出SM通道起始地址  

m_ppEcSlave[0]->m_pFmmu[0].m_nPhyStart = m_ppEcSlave[0]->  

    m_pSyncM[2*m_ppEcSlave[0]->m_nSlvType].m_nPhyStart;  

// 输出FMMU通道物理起始位等于0  

m_ppEcSlave[0]->m_pFmmu[0].m_cPhyStartBit = 0;  

// 输出FMMU通道类型为写操作  

m_ppEcSlave[0]->m_pFmmu[0].m_cType = 0x02;  

// 激活输出FMMU通道  

m_ppEcSlave[0]->m_pFmmu[0].m_cActive = 0x01;  

m_nOutSize = m_ppEcSlave[0]->m_pFmmu[0].m_nLength;  

//-----  

// 从站0输入FMMU配置  

//-----  

m_ppEcSlave[0]->m_pFmmu[1].m_nLgStart = 0x00001000;  

m_ppEcSlave[0]->m_pFmmu[1].m_nLength =  

    m_ppEcSlave[0]->m_nInLen;  

m_ppEcSlave[0]->m_pFmmu[1].m_cLgStartBit = 0;  

m_ppEcSlave[0]->m_pFmmu[1].m_cLgStopBit = 7;  

m_ppEcSlave[0]->m_pFmmu[1].m_nPhyStart = m_ppEcSlave[0]->  

    m_pSyncM[2*m_ppEcSlave[0]->m_nSlvType+ 1].m_nPhyStart;  

m_ppEcSlave[0]->m_pFmmu[1].m_cPhyStartBit = 0;

```

```

m_ppEcSlave[0]->m_pFmmu[1].m_cType = 0x01;
m_ppEcSlave[0]->m_pFmmu[1].m_cActive = 0x01;
m_nInSize = m_ppEcSlave[0]->m_pFmmu[1].m_nLength;
//-----//
//后续从站FMMU配置
//-----//
for (i=1;i<m_nEcSlave;i++)
{
    m_ppEcSlave[i]->m_nOutLen = m_ppEcSlave[i]->
        m_pSyncM[2*m_ppEcSlave[i]->m_nSlvType].m_nLength;
    m_ppEcSlave[i]->m_nOutOffset = m_ppEcSlave[i-1]->
        m_nOutOffset + m_ppEcSlave[i-1]->m_nOutLen;
    m_ppEcSlave[i]->m_pFmmu[0].m_nLgStart =
        m_ppEcSlave[i-1]->m_pFmmu[0].m_nLgStart
        + m_ppEcSlave[i-1]->m_nOutLen;
    m_ppEcSlave[i]->m_pFmmu[0].m_nLength =
        m_ppEcSlave[i]->m_nOutLen;
    m_ppEcSlave[i]->m_pFmmu[0].m_cLgStartBit = 0;
    m_ppEcSlave[i]->m_pFmmu[0].m_cLgStopBit = 7;
    m_ppEcSlave[i]->m_pFmmu[0].m_nPhyStart = m_ppEcSlave[i]->
        m_pSyncM[2*m_ppEcSlave[i]->m_nSlvType].m_nPhyStart;
    m_ppEcSlave[i]->m_pFmmu[0].m_cPhyStartBit = 0;
    m_ppEcSlave[i]->m_pFmmu[0].m_cType = 0x02;
    m_ppEcSlave[i]->m_pFmmu[0].m_cActive = 0x01;
    m_nOutSize = m_nOutSize +
        m_ppEcSlave[i]->m_pFmmu[0].m_nLength;
    m_ppEcSlave[i]->m_nInLen = m_ppEcSlave[i]->
        m_pSyncM[2*m_ppEcSlave[i]->m_nSlvType + 1].m_nLength;
    m_ppEcSlave[i]->m_nInOffset = m_ppEcSlave[i-1]->
        m_nInOffset + m_ppEcSlave[i-1]->m_nInLen;

    m_ppEcSlave[i]->m_pFmmu[1].m_nLgStart =
        m_ppEcSlave[i-1]->m_pFmmu[1].m_nLgStart
        + m_ppEcSlave[i-1]->m_nInLen;
    m_ppEcSlave[i]->m_pFmmu[1].m_nLength =
        m_ppEcSlave[i]->m_nInLen;
    m_ppEcSlave[i]->m_pFmmu[1].m_cLgStartBit = 0;
    m_ppEcSlave[i]->m_pFmmu[1].m_cLgStopBit = 7;
    m_ppEcSlave[i]->m_pFmmu[1].m_nPhyStart =
        m_ppEcSlave[i]->m_pSyncM[2*m_ppEcSlave[i]
        ->m_nSlvType+ 1].m_nPhyStart;
}

```

```

m_ppEcSlave[i]->m_pFmmu[1].m_cPhyStartBit = 0;
m_ppEcSlave[i]->m_pFmmu[1].m_cType = 0x01;
m_ppEcSlave[i]->m_pFmmu[1].m_cActive = 0x01;
m_nInSize = m_nInSize +
    m_ppEcSlave[i]->m_pFmmu[1].m_nLength;
}
}
}

```

6.4.4 状态机运行

主站执行状态机控制，完成所有从站设备的初始化，主站初始化过程及其相关寄存器操作如表 6.2 所列。其中对应的发送非周期性数据报文发送函数将在 6.4.5 中介绍。

表 6.2 初始化过程及其相关操作

初始化步骤	操作	使用命令	对应寄存器地址(长度)	对应发送非周期性数据报文函数
1	设置从站为初始化状态	APWR/BWR	0x0120 (2)	WriteAIControl (int nState)
复位进入“Init”状态				
2	写从站设置地址	APWR	0x10 (2)	WriteDIAddr()
3	配置邮箱通信 SM 通道	FPWR	0x800~0x80F	WriteSM(int i)
4	主站写状态控制寄存器， 请求“Pre-Op”状态	FPWR	0x0120 (2)	WriteAIControl (int nState)
5	主站读从站状态寄存器	FPRD	0x0130 (2)	ReadAIState()
进入“Pre-Op”状态，可以进行邮箱通信				
6	可选应用层邮箱数据通 信	FPWR/FPRD	0x1800,0x1C00	未使用
7	配置过程数据通信 SM 通道	FPWR	0x810~0x81F	WriteSM(int i)
8	配置 FMMU 通道 0 和 1	FPWR	0x600~0x61F	WriteFmmu(int i)
9	写状态控制寄存器，请求 “Safe-Op”状态	FPWR	0x0120 (2)	WriteAIControl (int nState)
10	主站读从站状态寄存器	FPRD	0x0130 (2)	ReadAIState()
进入“Safe-Op”状态，进行周期性数据通信，输出数据无效				
11	可选应用层邮箱数据通 信，配置应用参数	FPWR/FPRD	0x1800,0x1C00	未使用
12	主站写状态控制寄存器， 请求“Op”状态	FPWR	0x0120 (2)	WriteAIControl (int nState)
13	主站读从站状态寄存器	FPRD	0x0130 (2)	ReadAIState()
进入“Op”状态，输入和输出全部有效，仍然可以进行邮箱通信				

主站状态机处理函数的程序源代码如下：

```

-----  

// 主站状态机处理  

// int m_nStatus: 当前状态  

// int m_nRequire: 请求状态  

-----  

void CEcSimMaster::StateMachine()  

{  

    if (m_nRequire > m_nStatus)      // 状态上升  

    {  

        switch(m_nStatus)          // 从当前状态开始执行  

        {  

            case 10:                // 当前为Init状态  

                WriteAIControl(1);   // 请求Init状态  

                Delay(DELAYTIME);   // 延时  

                WriteSM(0);          // 配置从站SM0  

                Delay(DELAYTIME);   // 延时  

                WriteSM(1);          // 配置从站SM1  

                Delay(DELAYTIME);   // 延时  

                WriteDIAddr();       // 为从站配置站点地址  

                Delay(DELAYTIME);   // 延时  

                WriteAIControl(2);   // 请求Pre-Op状态  

                m_nStatus = 11;        // 主站程序进入中间状态  

                break;  

            case 11:  

                ReadAIState();        // 读从站状态  

                m_nStatus = 12;  

                break;  

            case 12:  

                if ((m_nReadStatus & 0x000f) == 2)  

                {  

                    // 检查从站是否进入Pre-Op状态  

                    m_nStatus = 20;        // 是，则主站程序进入Pre-Op状态  

                }  

                else  

                {  

                    m_nStatus = 11;        // 否则主站程序进入中间状态  

                }  

                break;  

            case 20:                  // 当前为Pre-Op状态  

                WriteSM(2);          // 配置从站SM2
        }
    }
}

```

```

    Delay(DELAYTIME);           // 延时
    WriteSM(3);                // 配置从站SM3
    Delay(DELAYTIME);           // 延时
    WriteFmmu(0);               // 配置从站FMMU0
    Delay(DELAYTIME);           // 延时
    WriteFmmu(1);               // 配置从站FMMU1
    Delay(DELAYTIME);           // 延时
    WriteAlControl(4);          // 请求Safe-Op状态
    m_nStatus = 21;              // 主站驱动程序进入等待状态
    break;

case 21:
    ReadAlState();             // 读从站状态
    m_nStatus = 22;
    break;

case 22:
    if ((m_nReadStatus & 0x000f) == 4)
    {
        // 检查从站是否进入Safe-Op状态
        m_nStatus = 40;          // 是，则主站程序进入Safe-Op状态
    }
    else
    {
        m_nStatus = 21;          // 否则退回上一状态继续等待
    }
    break;

case 40:                      // 当前为Safe-Op状态
    WriteAlControl(8);          // 请求Op状态
    m_nStatus = 41;              // 主站驱动程序进入等待状态
    break;

case 41:
    ReadAlState();             // 读从站状态
    m_nStatus = 42;
    break;

case 42:
    if ((m_nReadStatus & 0x000f) == 8)
    {
        // 检查从站是否进入Op状态
        m_nStatus = 80;          // 是，则主站程序进入Op状态
    }
    else
    {
        m_nStatus = 41;          // 否则退回上一状态继续等待
    }
}

```

```

    }
    break;
}
}

else if (m_nRequire < m_nStatus) // 状态下降
{
    WriteAlControl(m_nRequire / 10); // 请求从站进入请求的状态
    m_nStatus = m_nRequire;
}
}
}

```

6.4.5 发送非周期性 EtherCAT 数据报文

在初始化阶段，主站发送非周期性数据帧给从站，完成从站配置。需要配置的寄存器及其对应发送函数如表 6.2 所列。发送不同配置数据的非周期性数据帧各不相同，下面以主站对从站配置 SM 为例说明非周期性数据帧的发送。在函数中，主站使用 FPWR 命令写每个从站的 SM 配置寄存器 $0x800+8\times i$ ，其配置目标数据如表 6.1 所列。发送从站 SM 通道配置数据的函数程序源代码如下：

```

// -----
// 发送从站SM通道配置数据的非周期性数据帧
// int i: SM通道号
// -----
void CEcSimMaster::WriteSM(int i)
{
    ULONG slvCnt = 0;
    int frameLen = 0;

    ETHERNET_88A4_MAX_FRAME ethFrame;
    ethFrame.Ether.Destination = BroadcastEthernetAddress;
    ethFrame.Ether.Source = m_macAddr; // 数据帧源MAC地址
    ethFrame.Ether.FrameType = 0xA488; // 数据帧类型

    // 初始化一个子报文头
    ETYPES_EC_HEADER ecHdr;
    ecHdr.cmd = EC_CMD_TYPE_FPWR; // 使用FPWR命令
    ecHdr.idx = 0x82; // 数据帧索引号
    ecHdr.ado = 0x800 + i * 8; // 操作SM配置寄存器地址
    ecHdr.len = 8; // 子报文数据区长度为8
    ecHdr.next = 1; // 有后续子报文
}

```

```

ecHdr.res = 0;
ecHdr.irq = 0;

PBYTE pData = ethFrame.Data;
// 使用字节型指针指向数据帧的数据区

for (slvCnt = 0; slvCnt<m_nEcSlave; slvCnt++)
{
    if ((m_ppEcSlave[slvCnt]->m_pSyncM[i]).m_nLength != 0)
    {
        ecHdr.adp = m_ppEcSlave[slvCnt]->m_nDlAddr;
        // 写入从站设置地址
        memcpy(pData, &ecHdr, 10);      // 初始化子报文头
        pData = pData + 10;
        frameLen = frameLen + 10;
        memcpy(pData, &(m_ppEcSlave[slvCnt]->m_pSyncM[i]), 8);
        // 写入从站SM通道i的配置数据
        pData = pData + 8;
        frameLen = frameLen + 8;
        pData[0] = 0;
        pData[1] = 0;
        pData = pData + 2;
        frameLen = frameLen + 2;
    }
}

pData = pData - 13;
pData[0] = pData[0] & 0x7F;      // 清除最后一个子报文的后续报文标志

ethFrame.E88A4.Length = frameLen;
ethFrame.E88A4.Type = 1;
if(m_pNpfdev)
{
    m_pNpfdev->SendPacket(&ethFrame, frameLen + 16);
    // 发送以太网数据帧
    m_lSendFrame++;
}
}

```

6.4.6 发送周期性 EtherCAT 数据帧

EtherCAT通信中，主站使用写数据命令实现数据输出，使用读数据命令实现数据输入。主站周期性地发送包括写数据命令和读数据命令的数据帧，其数据帧格式相同。示例程序先准备好一个周期性数据帧框架，然后周期性地填入输出数据。发送数据帧后，数据帧经过每个从站时，从站将输入数据填入读数据命令的数据区，然后返回到主站。周期性数据帧可以使用FMMU逻辑寻址方式或设置寻址方式。本示例分别给出使用FMMU和设置寻址的周期性数据帧发送程序。

(1) 使用FMMU方式

分别使用LRD和LWR命令执行数据的输入和输出操作，构造一个数据帧框架，在周期性发送时向固定位置更新输出数据即可。准备数据帧框架的程序源代码如下：

```

// -----
// PrepareCyclicFrameFmmu(): 使用逻辑寻址准备周期性数据帧框架
// pFrame: 周期性数据帧的存放指针
// -----
void CEcSimMaster::PrepareCyclicFrameFmmu(unsigned char* pFrame)
{
    ULONG slvCnt = 0;
    int frameLen = 0;
    int outOffset = 0;
    PETYPE_EC_HEADER pEcHdr;

    ((PETHERNET_88A4_FRAME)pFrame)->Ether.Destination =
        BroadcastEthernetAddress; // 目的地为广播MAC地址
    ((PETHERNET_88A4_FRAME)pFrame)->Ether.Source = m_macAddr;
    ((PETHERNET_88A4_FRAME)pFrame)->Ether.FrameType =
        ETHERNET_FRAME_TYPE_ECAT; // 以太类型为0x88A4
    ((PETHERNET_88A4_FRAME)pFrame)->E88A4.Reserved = 0;
    ((PETHERNET_88A4_FRAME)pFrame)->E88A4.Type =
        ETTYPE_88A4_TYPE_ECAT; // 数据类型为ECAT
    // 初始长度为最大数据长度
    ((PETHERNET_88A4_FRAME)pFrame)->E88A4.Length =
        ETHERNET_MAX_FRAME_LEN-ETHERNET_88A4_FRAME_LEN;
    // 数据全部置零
    memset(pFrame+ETHERNET_88A4_FRAME_LEN, 0x00,
           ETHERNET_MAX_FRAME_LEN-ETHERNET_88A4_FRAME_LEN);

    // -----
    // 使用LWR命令写从站ESC存储区以实现数据的输出
    // 使用EtherCAT子报文类型指针操作数据区

```

```

// -----
pEcHdr = (PETYPE_EC_HEADER)
    (pFrame+ETHERNET_88A4_FRAME_LEN+frameLen);

ecHdr.cmd = EC_CMD_TYPE_LWR;
ecHdr.idx = 0;
ecHdr.laddr = m_ppEcSlave[0]->m_pFmmu[0].m_nLgStart;
ecHdr.len = m_nOutSize;
ecHdr.next = 1;
ecHdr.res = 0;
ecHdr.irq = 0;
frameLen = ETYPE_EC_HEADER_LEN + ecHdr.len + 2;

// -----
// 使用LRD命令读从站ESC存储区以实现数据的输入
// 使用EtherCAT子报文类型指针操作数据区
// -----
pEcHdr = (PETYPE_EC_HEADER)
    (pFrame+ETHERNET_88A4_FRAME_LEN+frameLen);

ecHdr.cmd = EC_CMD_TYPE_LRD;
ecHdr.idx = 0;
ecHdr.laddr = m_ppEcSlave[0]->m_pFmmu[1].m_nLgStart;
ecHdr.len = m_nInSize;
ecHdr.next = 1;
ecHdr.res = 0;
ecHdr.irq = 0;
frameLen = ETYPE_EC_HEADER_LEN + ecHdr.len + 2;

// -----
// 使用BRD命令读所有从站的状态寄存器，对所有从站做实时监测
// 使用EtherCAT子报文类型指针操作数据区
// -----
pEcHdr = (PETYPE_EC_HEADER)
    (pFrame+ETHERNET_88A4_FRAME_LEN+frameLen);

ecHdr.cmd = EC_CMD_TYPE_BRD;
ecHdr.adp = 0;
ecHdr.ado = 0x120;
ecHdr.len = 2;
ecHdr.next = 0;
frameLen = ETYPE_EC_HEADER_LEN + ecHdr.len + 2

```

```

// 设置实际数据长度
((PETHERNET_88A4_FRAME)pFrame)->E88A4.Length = frameLen;
}

```

周期性发送数据帧时，把 m_OutputImage 中更新后的输出数据复制到数据帧框架的 LWR 命令数据区，然后将其发送，该过程的程序源代码如下：

```

//-----  

//SendCyclicFrameFmmu(): 发送周期性数据帧，使用FMMU  

//pFrame: 周期性数据帧存放指针  

//-----  

void CEcSimMaster::SendCyclicFrameFmmu(unsigned char* pFrame)  

{  

    ULONG slvCnt = 0;  

    int frameLen = 0;  

    int outOffset = 0;  

    PETYPE_EC_HEADER pEcHdr;  

    PVOID pData;  

    // ----- 复制输出数据 -----  

    pEcHdr = (PETYPE_EC_HEADER)  

        (pFrame+ETHERNET_88A4_FRAME_LEN+frameLen);  

    pEcHdr.idx = m_nIndex;  

    pData = ENDOF(ecHdr);  

    memcpy(pData, m_OutputImage, ecHdr.len); // 复制输出数据  

    // ----- 发送数据帧 -----  

    if(m_pNpfdev)  

    {  

        m_pNpfdev->SendPacket(&pFrame,  

            (ETHERNET_88A4_FRAME_LEN +  

            ((PETHERNET_88A4_FRAME)pFrame)->E88A4.Length));  

        m_lSendFrame++;  

    }  

    m_nIndex++; // 周期性数据帧索引增加  

    if(m_nIndex>=0x80) // 周期性数据帧索引从0~0x7F  

    {  

        m_nIndex = 0;  

    }
}

```

(2) 使用设置寻址方式

使用 FPRD 和 FPWR 命令操作站点的输入和输出。首先构造一个数据帧框架，在周期性发送数据时向固定位置更新输出数据即可。准备数据帧框架的程序源代码如下：

```

// -----
// PrepareCyclicFrame(): 准备设置寻址周期性数据帧框架
// pFrame: 周期性数据帧的存放指针
// -----
void CEcSimMaster::PrepareCyclicFrame(unsigned char* pFrame)
{
    ULONG slvCnt = 0;
    int frameLen = 0;
    int outOffset = 0;
    PETYPE_EC_HEADER pEcHdr;

    ((PETHERNET_88A4_FRAME)pFrame)->Ether.Destination =
        BroadcastEthernetAddress; // 目的地为广播MAC地址
    ((PETHERNET_88A4_FRAME)pFrame)->Ether.Source = m_macAddr;
    ((PETHERNET_88A4_FRAME)pFrame)->Ether.FrameType =
        ETHERNET_FRAME_TYPE_ECAT; // 以太类型为0x88A4
    ((PETHERNET_88A4_FRAME)pFrame)->E88A4.Reserved = 0;
    ((PETHERNET_88A4_FRAME)pFrame)->E88A4.Type =
        ETTYPE_88A4_TYPE_ECAT; // 数据类型为ECAT
    // 初始长度为最大数据长度
    ((PETHERNET_88A4_FRAME)pFrame)->E88A4.Length =
    ETHERNET_MAX_FRAME_LEN-ETHERNET_88A4_FRAME_LEN;
    // 数据全部置零
    memset(pFrame+ETHERNET_88A4_FRAME_LEN, 0x00,
           ETHERNET_MAX_FRAME_LEN-ETHERNET_88A4_FRAME_LEN);

    // -----
    // 使用FPWR命令完成输出数据，每个从站使用一个子报文
    // 使用EtherCAT子报文类型指针操作数据区
    // -----
    for (slvCnt = 0; slvCnt<m_nEcSlave; slvCnt++)
    {
        pEcHdr = (PETYPE_EC_HEADER)
            (pFrame+ETHERNET_88A4_FRAME_LEN+frameLen);
        ecHdr.cmd = EC_CMD_TYPE_FPWR;
        ecHdr.idx = 0;
        ecHdr.adp = m_ppEcSlave[slvCnt]->m_nDlAddr;
    }
}

```

```

ecHdr.ado = m_ppEcSlave[slvCnt]->
    m_pSyncM[m_ppEcSlave[slvCnt]->m_nSlvType * 2].m_nPhyStart;
                                // 子报文起始地址为输出SM起始地址
ecHdr.len = m_ppEcSlave[slvCnt]->
    m_pSyncM[m_ppEcSlave[slvCnt]->m_nSlvType * 2].m_nLength;
                                // 数据长度为输出SM数据长度
ecHdr.next = 1;
ecHdr.res = 0;
ecHdr.irq = 0;
frameLen = frameLen + ETYPE_EC_HEADER_LEN + ecHdr.len + 2;
}

// -----
// 使用FPRD命令完成读输入数据，每个从站使用一个子报文
// 使用EtherCAT子报文类型指针操作数据区
// -----
for (slvCnt = 0; slvCnt<m_nEcSlave; slvCnt++)
{
    pEcHdr = (PETYPE_EC_HEADER)
        (pFrame+ETHERNET_88A4_FRAME_LEN+frameLen);
    ecHdr.cmd = EC_CMD_TYPE_FPRD;
    ecHdr.idx = 0;
    ecHdr.adp = m_ppEcSlave[slvCnt]->m_nDlAddr;
    ecHdr.ado = m_ppEcSlave[slvCnt]->
        m_pSyncM[m_ppEcSlave[slvCnt]->m_nSlvType * 2 +1].m_nPhyStart;
                                // 子报文起始地址为输入SM起始地址
    ecHdr.len = m_ppEcSlave[slvCnt]->
        m_pSyncM[m_ppEcSlave[slvCnt]->m_nSlvType * 2 +1].
        m_nLength; // 数据长度为输入SM数据长度
    ecHdr.next = 1;
    ecHdr.res = 0;
    ecHdr.irq = 0;
    frameLen = frameLen + ETYPE_EC_HEADER_LEN + ecHdr.len + 2;
}

// -----
// 使用BRD命令读所有从站的状态寄存器，对所有从站做实时监测
// 使用EtherCAT子报文类型指针操作数据区
// -----
pEcHdr = (PETYPE_EC_HEADER)
    (pFrame+ETHERNET_88A4_FRAME_LEN+frameLen);

```

```

    ecHdr.cmd = EC_CMD_TYPE_BRD;
    ecHdr.adp = 0;
    ecHdr.ado = 0x120;
    ecHdr.len = 2;
    ecHdr.next = 0;
    frameLen = frameLen + ETYPE_EC_HEADER_LEN + ecHdr.len + 2
    // 设置实际数据长度
    ((PETHERNET_88A4_FRAME)pFrame)->E88A4.Length = frameLen;
}

```

周期性发送数据帧时，把 m_OutputImage 中的输出数据分别复制到数据帧框架的 FPWR 命令数据区，然后将其发送，该过程的程序源代码如下：

```

//-----  

//SendCyclicFrame (): 发送周期性数据帧，不使用FMMU  

//pFrame: 周期性数据帧的存放指针  

//-----  

void CEcSimMaster::SendCyclicFrame(unsigned char* pFrame)  

{  

    ULONG slvCnt = 0;  

    int frameLen = 0;  

    int outOffset = 0;  

    int index;  

    PETYPE_EC_HEADER pEcHdr;  

    PVOID pData;  

    index = m_nIndex;  

    for (slvCnt = 0; slvCnt < m_nEcSlave; slvCnt++)  

    {  

        pEcHdr = (PETYPE_EC_HEADER)  

            (pFrame+ETHERNET_88A4_FRAME_LEN+frameLen);  

        ecHdr.idx = index;  

        pData = ENDOF(ecHdr);  

        // 复制输出数据  

        memcpy(pData, m_OutputImage + outOffset, ecHdr.len);  

        outOffset = outOffset + ecHdr.len;  

        frameLen = ETYPE_EC_HEADER_LEN + ecHdr.len + 2;  

        index = 0;  

    }  

    // 发送数据帧  

    if(m_pNpfdev)  

    {

```

```

    m_pNpfdev->SendPacket(&ethFrame,
                           (ETHERNET_88A4_FRAME_LEN +
                            ((PETHERNET_88A4_FRAME)pFrame)->E88A4.Length);
    m_lSendFrame++;
}

m_nIndex++;
if(m_nIndex>=0x80)
{
    m_nIndex = 0;
}
}

```

6.4.7 接收 EtherCAT 数据帧

主站检查所有的返回数据帧，得到非周期性操作结果或周期性输入数据。程序中首先判断数据帧类型是否为 0x88A4 来挑选 EtherCAT 数据帧，然后使用数据帧中的第一个子报文的索引来区分周期性数据帧和非周期性数据帧，索引号小于 0x80 的报文为周期性报文，大于等于 0x80 的报文为非周期性报文。周期性数据写入到数组 m_InputImage 中，非周期性数据报文被立即处理。根据子报文索引号可以判断非周期性数据报文的类型，随后做出相应的处理。接收数据帧的程序源代码如下：

```

// -----
//CheckFrames(): 检查接收到的数据帧
//-----

long CEcSimMaster::CheckFrames()
{
    PETHERNET_88A4_MAX_FRAME ethFrame;
    BYTE packetData[sizeof(ETHERNET_88A4_MAX_FRAME)];
    PUSHORT pFrameType;
    int nDataNumber = 0;
    // 从NPF驱动程序得到一个数据帧, nData为数据帧长度
    long nData = m_pNpfdev->CheckRecvFrame(packetData);
    //用指针ethFrame指到所收到的数据帧
    ethFrame = PETHERNET_88A4_MAX_FRAME(packetData);

    while(nData != 0)
    {
        if (nData < ETHERNET_FRAME_LEN) // 如果数据帧不完整
            return FALSE;

        pFrameType = FRAMETYPE_PTR(ethFrame); // 得到数据帧类型

```

```

if (*pFrameType == ETHERNET_FRAME_TYPE_ECAT)
{
    // 如果是EtherCAT数据帧类型0x88A4
    m_lRecvFrame++; // 接收计数将增加
    PETHERNET_88A4_MAX_HEADER p88A4 =
        (PETHERNET_88A4_MAX_HEADER) ENDOF(pFrameType);

    if (p88A4->E88A4.Type == ETYPE_88A4_TYPE_ECAT)
    { //EtherCAT 数据帧
        ULONG e88A4Len = p88A4->E88A4.Length;
        // pHHead指向第一个EtherCAT命令报文
        PETYPE_EC_HEADER pHHead = &p88A4->FirstEcHead;
        nDataNumber = 0;
        if (pHead->idx < EC_HEAD_IDX_SLAVECMD)
        { // 是周期性数据帧
            while (pHead)
            { // 遍历每一个EtherCAT子报文
                if (e88A4Len < ETYPE_EC_CMD_LEN(pHead))
                    break; // 数据不完整

                if (pHead->cmd == EC_CMD_TYPE_FPWR)
                { // 是FPWR命令子报文

                }
                else if (pHead->cmd == EC_CMD_TYPE_FPRD)
                { // 是FPRD命令子报文, 将输入数据复制到输入映像区
                    memcpy(m_InputImage + nDataNumber,
                           ENDOF(pHead), pHHead->len);
                    nDataNumber = nDataNumber + pHHead->len;
                }

                else if (pHead->cmd == EC_CMD_TYPE_BRD)
                { // 是BRD命令子报文

                }
                else if (pHead->cmd == EC_CMD_TYPE_LRD)
                { // 是LRD命令子报文, 将输入数据复制到输入映像区
                    memcpy(m_InputImage, ENDOF(pHead),
                           m_nInSize);
                }
            }
            e88A4Len -= ETYPE_EC_CMD_LEN(pHead);
        }
    }
}

```

```

        if ( pHead->next )// 有后续命令子报文
            pHead = NEXT_EcHeader(pHead);
        else
            pHead = NULL;
    }
}
else // 非周期性数据帧, 用Index表示命令操作类型
{
    if (pHead->idx == 0x80)
    { // 读从站状态命令
        while ( pHead )
        { // look for each EtherCAT sub telegram
            if ( e88A4Len < ETTYPE_EC_CMD_LEN(pHead) )
                break;

            if (pHead->cmd == EC_CMD_TYPE_BRD)
            {
                USHORT * nSt = (USHORT *)ENDOF(pHead);
                m_nReadStatus = * nSt;
            }
            e88A4Len -= ETTYPE_EC_CMD_LEN(pHead);

            if ( pHead->next )
                pHead = NEXT_EcHeader(pHead);
            else
                pHead = NULL;
        }
    }
}

// 从NPF驱动程序读下一个数据帧
nData = m_pNpfdev->CheckRecvFrame(packetData);
ethFrame = PETHERNET_88A4_MAX_FRAME(packetData);
}

return TRUE;
}

```

6.5 主站实例程序

以下是一个主站实例程序，PC 计算机作为主站，控制一个微处理器控制从站和一个 I/O 从站，微处理器从站使用 AVR 处理器，如图 6.5 所示。

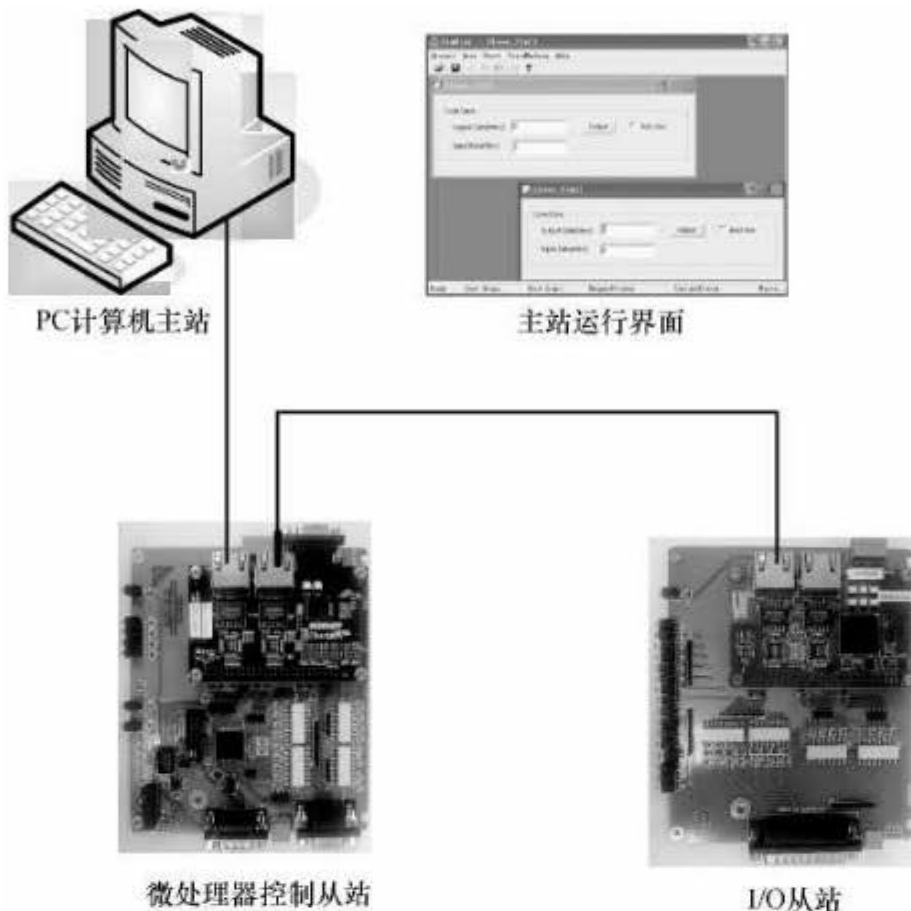


图 6.5 主站实例程序硬件配置

主站实例程序的运行包括通信配置初始化和周期性运行两个任务：

(1) 配置通信参数，初始化主站类，并启动多媒体定时器

程序中定义了一个主站类对象 `m_pEcMaster`，所有的 EtherCAT 通信都使用这个数据对象实现。程序首先初始化主站类对象 `m_pEcMaster`，接着启动多媒体定时器，其运行流程如图 6.6 所示。

图中，`OnFileNewproject()` 函数用以实现实例程序的初始化，其详细内容会在 6.5.1 中介绍；`OnStart()` 函数用以启动多媒体定时器，其具体方法参见 Windows 编程相关资料 [30]，本书不做详细介绍。

(2) 周期性运行控制

该过程的运行流程如图 6.7 所示，其任务包括：

- 发送周期性数据帧，检查接收数据帧；
- 查询执行非周期性通信操作，完成状态机控制；
- 周期性刷新界面显示。

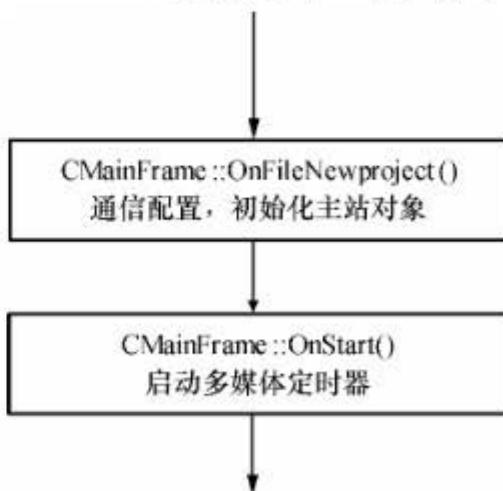


图 6.6 主站实例程序初始化运行流程

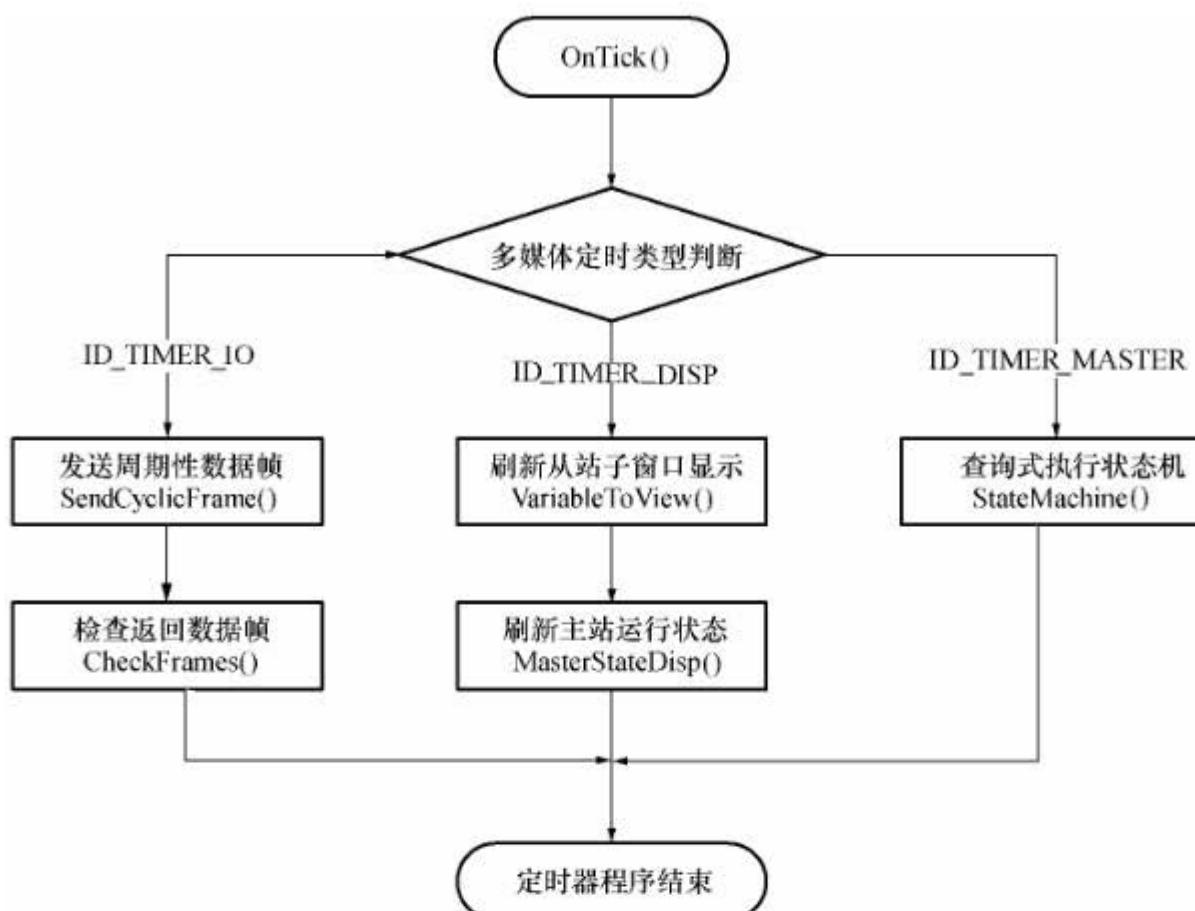


图 6.7 主站实例程序周期性运行流程

6.5.1 通信配置初始化流程

主站实例程序使用多窗口界面，用一个子窗口表示一个从站，如图 6.8 所示。程序运行后，单击 Project→New Project 菜单，则弹出主站配置界面（如图 6.9 所示），它可以对主站和从站进行配置。



图 6.8 主站实例程序的运行界面

- (1) 主站配置区为图 6.9 中“Master Configuration”部分，其配置项包括：

- Ethernet Adapter 网卡选择;
- SlaveCnt 从站数目;
- CycleTime 通信周期;
- Use FMMU 周期性通信使用 FMMU 寻址或设置寻址。

(2) 从站配置区为图 6.9 中“Slave Configuration”部分，其配置方式有如下两种情况：

- 从站周期性通信所使用的 SM 通道参数，从站邮箱通信的 SM 在程序中进行默认配置（参见表 6.1）；
- IO 从站的周期性通信 SM 配置固定，而且不支持邮箱通信，所以在配置界面中直接给出正确配置（选择 IO-16 IN/16 OUT）。

以上配置完成后，单击 Dialog 对话框的“OK”按钮，新建一个主站工程，进入主运行界面。

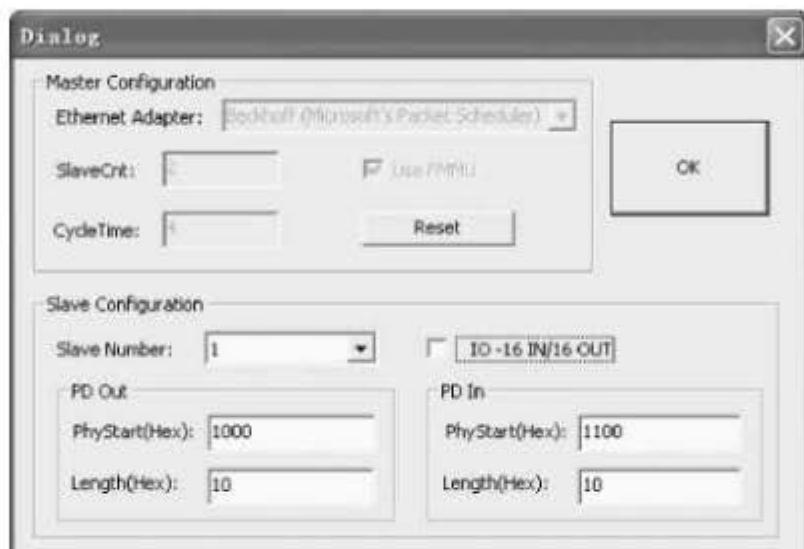


图 6.9 主站实例程序配置界面

实现主站初始化配置的程序源代码如下：

```

//-----
//OnFileNewproject(): 新建一个主站工程，配置通信参数
//-----
void CMainFrame::OnFileNewproject()
{
    int i;

    if(dlg.DoModal() == IDOK) // DoModal() 函数弹出配置界面
    {
        // 如果在配置界面上单击“OK”按钮
        // 新建一个主站类数据对象，将其指针赋予m_pEcMaster
        m_pEcMaster = new CEcSimMaster(dlg.m_nSlvCnt); // 见6.4.2
        // 从配置界面得到通信周期
        m_pEcMaster->m_nCycTime = dlg.m_nCycleTime;
        // 从配置界面得到选用网卡的MAC地址
        m_pEcMaster->m_macAddr= dlg.macAddr;
        // 从配置界面得到是否使用FMMU机制
    }
}

```

```

m_pEcMaster->m_bFmmu= dlg.m_bFmmu;
// 从配置界面得到从站数目
m_pEcMaster->m_nEth= dlg.ethNum;
for (i=0;i<m_pEcMaster->m_nEcSlave;i++)
{
    if ((dlg.m_ppSmConfig[i])->m_nSlvType == ISASLAVEFRM)
        { // 如果从站为微处理器控制从站类型，重新配置SM参数
        m_pEcMaster->CreateSlave(i,ISASLAVEFRM,
        dlg.m_ppSmConfig[i]->m_nPhyStart1,
        dlg.m_ppSmConfig[i]->m_nLength1,
        dlg.m_ppSmConfig[i]->m_nPhyStart2,
        dlg.m_ppSmConfig[i]->m_nLength2); // 见6.4.3节
    }
}

// 准备周期性数据帧框架
if (m_pEcMaster->m_bFmmu == 1)
{
    // 调用ImageAssign()函数，配置FMMU
    m_pEcMaster->ImageAssign(); // 见6.4.3节
    // 使用FMMU准备周期性数据帧框架
    m_pEcMaster->PrepareCyclicFrameFmmu(); // 见6.4.6节的(1)部分
}
else
{
    // 使用设置寻址准备周期性数据帧框架
    m_pEcMaster->PrepareCyclicFrame(); // 见6.4.6节的(2)部分
}

// 启动主站运行，初始化m_pNpfdev数据对象，打开网卡通信
if (!(m_pEcMaster->Open())) // 见6.4.2节
{
    // 函数执行失败，弹出信息提示框，并返回
    MessageBox("Open Device Error!");
    return;
}

POSITION ps = NULL;
CMultiDocTemplate * m_pDocTemplate;
CSimEcatDoc * m_pDocument;
theApp.CloseAllDocuments(0);
// 按照从站数目新建子窗口

```

```

for (i=0;i<m_pEcMaster->m_nEcSlave;i++)
{
    SendMessage(WM_COMMAND, ID_FILE_NEW);
}

CString str;
CString sType[2];
sType[0] = "(IO)";
sType[1] = "(uC)";

ps = theApp.GetFirstDocTemplatePosition();
m_pDocTemplate = (CMultiDocTemplate *)
                    (theApp.GetNextDocTemplate(ps));
ps = m_pDocTemplate->GetFirstDocPosition();
// 设置每个从站子窗口标题
for(int j=0;j<m_pEcMaster->m_nEcSlave;j++)
{
    m_pDocument = (CSimEcatDoc *)
                    (m_pDocTemplate->GetNextDoc(ps));
    str.Format("Slave_%d",j+1);
    str = str +
        sType[m_pEcMaster->m_ppEcSlave[j]->m_nSlvType];
    m_pDocument->SetTitle(str);
}
m_bTimerStarted = FALSE;
}
}

```

6.5.2 周期性运行控制

实例程序中使用多媒体定时器完成周期性数据通信和轮询非周期性任务，如状态机处理、界面刷新等。在该程序中新建了三个定时器，使用不同的标识符表示：

(1) ID_TIMER_IO 周期性数据通信定时器，优先级最高，周期为前面所配置的通信周期时间；

(2) ID_TIMER_MASTER 非周期任务查询定时器，优先级较低，周期为 50 ms；

(3) ID_TIMER_DISP 刷新界面显示定时器，优先级最低，周期为 100 ms。

多媒体定时器响应的程序源代码如下：

```

void CMainFrame::OnTick(UINT nId, CTimer *pTimer)
{
    int i = 0;

```

```

int state = 0;
long sent = 0;
long recvd = 0;

switch( nId )
{
    case ID_TIMER_MASTER:           // 非周期性数据通信控制
        if(m_pEcMaster)
        {
            m_pEcMaster->StateMachine(); // 执行状态机函数, 见6.4.4节
        }
        break;

    case ID_TIMER_IO:              // 周期性数据通信控制
        if (m_pEcMaster->m_nStatus >=40)
        {
            if (m_pEcMaster->m_bFmmu == 1)
            {      // 使用FMMU发送周期性数据通信
                m_pEcMaster->SendCyclicFrameFmmu(); // 见6.4.6节的(1)部分
            }
            else
            {      // 使用设置寻址发送周期性数据通信
                m_pEcMaster->SendCyclicFrame(); // 见6.4.6节的(2)部分
            }
        }
        m_pEcMaster->CheckFrames();      // 检查返回数据帧, 见6.4.7节
        break;

    case ID_TIMER_DISP:
        VariableToView();             // 刷新从站子窗口数据, 源代码略
        MasterStateDisp();            // 刷新主站运行状态, 源代码略
        break;
}
}

```

第7章 从站驱动程序

本章介绍基于第4章中Atmega128单片机控制的EtherCAT从站驱动程序，使用标准C语言开发，可以方便地移植到其他嵌入式控制平台中。

7.1 从站驱动程序头文件 ec_def.h

从站驱动程序头文件主要定义了重要的常量、数据结构及全局变量，其中一些常量和数据结构将在后续章节中进一步介绍。

从站驱动程序头文件ec-def.h的程序源代码如下：

```
//=====
// 创建:          2008/01/15
// 文件名:        EC_DEF
// 文件类型:      C
// 目的:          ESC 基本定义
//=====

//-----基本数据类型定义-----
#define UINT8     unsigned char
#define UINT16    unsigned int
#define UINT32    unsigned long
#define INT8      char
#define INT16     int
#define INT32     long
#define UCHAR     unsigned char
#define BOOL      unsigned char
#define TRUE      1
#define FALSE     0

// ESC基地址
#define ESC_REG_ENTRY 0x2000

//-----常量定义-----
//-----协议相关变量定义，主要为ProcessData与Mailbox
#define MAX_RX_PDOS      0x0001
```

```

#define MAX_TX_PDOS          0x0001
#define MIN_PD_WRITE_ADDRESS  0x1000
#define MAX_PD_WRITE_ADDRESS  0x2000
#define MIN_PD_READ_ADDRESS   0x1000
#define MAX_PD_READ_ADDRESS   0x2000
#define NO_OF_PD_INPUT_BUFFER 0x0003
#define NO_OF_PD_OUTPUT_BUFFER 0x0003

#define MAX_PD_INPUT_SIZE     0x0040
#define MAX_PD_OUTPUT_SIZE    0x0040
#define MAX_MB_INPUT_SIZE     0x0040
#define MAX_MB_OUTPUT_SIZE    0x0040
#define MIN_MBX_SIZE          0x0020
#define MAX_MBX_SIZE          0x0400
#define MIN_MBX_WRITE_ADDRESS 0x1000
#define MIN_MBX_READ_ADDRESS   0x1000
#define MAX_MBX_WRITE_ADDRESS 0x2000
#define MAX_MBX_READ_ADDRESS   0x2000

// 状态机相关定义
#define STATE_INIT      ((UINT8)0x01)
#define STATE_PREOP     ((UINT8)0x02)
#define STATE_BOOT      ((UINT8)0x03)
#define STATE_SAFEOP    ((UINT8)0x04)
#define STATE_OP        ((UINT8)0x08)

#define STATE_MASK      ((UINT8)0x0F)
#define STATE_CHANGE    ((UINT8)0x10)
#define STATE_ERRACK    ((UINT8)0x10)
#define STATE_ERROR     ((UINT8)0x10)

#define INIT_2_INIT     ((STATE_INIT << 4) | STATE_INIT)
#define INIT_2_PREOP    ((STATE_INIT << 4) | STATE_PREOP)
#define INIT_2_SAFEOP   ((STATE_INIT << 4) | STATE_SAFEOP)
#define INIT_2_OP        ((STATE_INIT << 4) | STATE_OP)

#define PREOP_2_INIT    ((STATE_PREOP << 4) | STATE_INIT)
#define PREOP_2_PREOP   ((STATE_PREOP << 4) | STATE_PREOP)
#define PREOP_2_SAFEOP  ((STATE_PREOP << 4) | STATE_SAFEOP)
#define PREOP_2_OP       ((STATE_PREOP << 4) | STATE_OP)

```

```

#define SAFEOP_2_INIT    ((STATE_SAFEOP << 4) | STATE_INIT)
#define SAFEOP_2_PREOP   ((STATE_SAFEOP << 4) | STATE_PREOP)
#define SAFEOP_2_SAFEOP((STATE_SAFEOP << 4) | STATE_SAFEOP)
#define SAFEOP_2_OP      ((STATE_SAFEOP << 4) | STATE_OP)

#define OP_2_INIT        ((STATE_OP << 4) | STATE_INIT)
#define OP_2_PREOP       ((STATE_OP << 4) | STATE_PREOP)
#define OP_2_SAFEOP      ((STATE_OP << 4) | STATE_SAFEOP)
#define OP_2_OP          ((STATE_OP << 4) | STATE_OP)

// SM通道定义
#define MAILBOX_WRITE     0
#define MAILBOX_READ      1
#define PROCESS_DATA_OUT  2
#define PROCESS_DATA_IN   3

// 相关中断定义，寄存器0x220~0x221位判断
#define AL_CONTROL_EVENT ((UINT16) 0x0001)
#define SYNC0_EVENT       ((UINT16) 0x0400)
#define SYNC1_EVENT       ((UINT16) 0x0800)
#define SM_CHANGE_EVENT   ((UINT16) 0x0010)

#define MAILBOX_WRITE_EVENT ((UINT16) 0x0100)
#define MAILBOX_READ_EVENT ((UINT16) 0x0200)
#define PROCESS_OUTPUT_EVENT ((UINT16) 0x0400)
#define PROCESS_INPUT_EVENT ((UINT16) 0x0800)

// AL状态码，写入寄存器0x134~0x135
#define ALSTATUSCODE_NOERROR           0x0000
#define ALSTATUSCODE_UNSPECIFIEDERROR  0x0001
#define ALSTATUSCODE_INVALIDALCONTROL  0x0011
#define ALSTATUSCODE_UNKNOWNALCONTROL  0x0012
#define ALSTATUSCODE_BOOTNOTSUPP       0x0013
#define ALSTATUSCODE_NOVALIDFIRMWARE   0x0014
#define ALSTATUSCODE_INVALIDMBXCFGINBOOT 0x0015
#define ALSTATUSCODE_INVALIDMBXCFGINPRE 0x0016
#define ALSTATUSCODE_INVALIDSMCFG      0x0017
#define ALSTATUSCODE_NOVALIDINPUTS     0x0018
#define ALSTATUSCODE_NOVALIDOUTPUTS    0x0019
#define ALSTATUSCODE_SYNCERROR         0x001A

```

```

#define ALSTATUSCODE_SMWATCHDOG           0x001B
#define ALSTATUSCODE_SYNCTYPESNOTCOMPATIBLE 0x001C
#define ALSTATUSCODE_INVALIDSMOUTCFG      0x001D
#define ALSTATUSCODE_INVALIDSMINCFG      0x001E

#define ALSTATUSCODE_WAITFORCOLDSTART    0x0020
#define ALSTATUSCODE_WAITFORINIT        0x0021
#define ALSTATUSCODE_WAITFORPREOP      0x0022
#define ALSTATUSCODE_WAITFORSAFEOP     0x0023
#define ALSTATUSCODE_DCINVALIDDSYNCCFG  0x0030
#define NOERROR_NOSTATECHANGE         0xFE
#define NOERROR_INWORK                0xFF

// 配置时出错标志代码
#define SYNCMANCHADDRESS               0x01
#define SYNCMANCHSETTINGS              0x03
#define SYNCMANCHSIZE                 0x02

//Sync Manager寄存器位定义
#define SM_PDINITMASK                0x0D
#define SM_TOGGLEMASTER               0x02
#define SM_ECATENABLE                 0x01
#define SM_INITMASK                  0x0F
#define ONE_BUFFER                    0x02
#define THREE_BUFFER                  0x00
#define PD_OUT_BUFFER_TYPE            THREE_BUFFER
#define PD_IN_BUFFER_TYPE             THREE_BUFFER
#define SM_WRITESETTINGS              0x04
#define SM_READSETTINGS               0x00
#define SM_PDIDISABLE                 0x01
#define WATCHDOG_TRIGGER              0x40

-----  

// ESC寄存器结构体定义
-----  

// AL中断事件寄存器定义, 0x220~0x223
typedef struct
{
    UINT8     Byte[4];
} UALEVENT;

```

```

// AL中断屏蔽寄存器定义, 0x204~0x207
typedef struct
{
    UINT16 Word[2];
} UALEMENTMASK; // 0x204

// SM结构体定义
typedef struct
{
    UINT16 sm_physical_addr;           // SM物理起始地址
    UINT16 sm_length;                 // SM长度
    UINT8 sm_register_control;
    UINT8 sm_register_status;
    UINT8 sm_register_activate;
    UINT8 sm_register_pdictl;
} TSYNCFMAN;

// EEPROM操作结构体定义
typedef struct
{
    UINT8 eeprom_config;
    UINT8 eeprom_pdi_acstate;
    UINT16 eeprom_ctl_status;
    UINT32 eeprom_addr;
    UINT32 eeprom_data[2];
} TEEPROM_DEF;

// MII操作结构体定义
typedef struct
{
    UINT16 mii_ctl_status;
    UINT8 mii_phy_addr;
    UINT8 mii_phy_registeraddr;
    UINT16 mii_phy_data;
} TMII;

// FMMU结构体定义
typedef struct
{

```

```

    UINT32 logical_start_addr;
    UINT16 length;
    UINT8 logical_start_bit;
    UINT8 logical_stop_bit;
    UINT16 physical_start_addr;
    UINT8 physical_start_bit;
    UINT8 type;
    UINT8 activate;
    UINT8 res[3];
}TFMMU;

// 分步式时钟结构体定义
typedef struct
{
    UINT32 receive_port[4];
    UINT32 sys_time[2];
    UINT8 receive_time_pu[8];
    UINT32 sys_time_offset[2];
    UINT32 sys_time_delay;
    UINT32 sys_time_diff;
    UINT16 speed_cnt_start;
    UINT16 speed_cnt_diff;
    UINT8 sys_filter_depth;
    UINT16 res27[37];
    UINT8 cyclic_unit_ctl;
    UINT8 activation;
    UINT16 pulse_length;
    UINT16 res28[5];
    UINT8 sync0_status;
    UINT8 sync1_status;
    UINT32 start_time_cyclic[2];
    UINT32 next_sync1_pulse[2];
    UINT32 sync0_cyclic_time;
    UINT32 sync1_cyclic_time;
    UINT8 latch0_ctl;
    UINT8 latch1_ctl;
    UINT16 res29[2];
    UINT8 latch0_status;
    UINT8 latch1_status;
    UINT32 latch0_time_pedge[2];
}

```

```

    UINT32 latch0_time_nedge[2];
    UINT32 latch1_time_pedge[2];
    UINT32 latch1_time_nedge[2];
    UINT16 res30[16];
    UINT32 ecat_bchangee_time;
    UINT16 res31[17];
    UINT32 pdi_bstarte_time;
    UINT32 pdi_bchangee_time;
}TDC;

```

驱动程序定义了一个与 ESC 内部寄存器分布相对应的结构体数据类型 TESC_REG，结构体数据类型中的变量与 ESC 中的寄存器一一对应。以下列出 TESC-REG 结构体定义的程序源代码。

```

// ESC寄存器整体结构体定义
typedef struct
{
    UINT8 type;                                //0x0000
    UINT8 revision;                            //0x0001
    UINT16 build;                             //0x0002
    UINT8 fmmus_supported;                    //0x0004
    UINT8 sm_supported;                      //0x0005
    UINT8 ram_size;                           //0x0006
    UINT8 port_descriptor;                   //0x0007
    UINT16 esc_feature;                     //0x0008
    UINT16 res1[3];
    UINT16 station_addr;                    //0x0010
    UINT16 alias_addr;                      //0x0012
    UINT16 res2[6];
    UINT8 write_enable;                     //0x0020
    UINT8 write_protection;                 //0x0021
    UINT16 res3[7];
    UINT8 esc_wrenable;                    //0x0030
    UINT8 esc_wrprotection;                //0x0031
    UINT16 res4[7];
    UINT8 esc_reset;                        //0x0040
    UINT8 res5[191];
    UINT32 esc_dlctl;                      //0x0100
    UINT16 res6[2];
    UINT16 physical_rdwr_offset;           //0x0108
    UINT16 res7[3];
    UINT16 esc_dlstatus;                  //0x0110
}

```

```

UINT16 res8[7];
UINT16 al_ctl; //0x0120
UINT16 res9[7];
UINT16 al_status; //0x0130
UINT16 res10;
UINT16 al_statuscode; //0x0134
UINT16 res11[5];
UINT16 pdi_ctl; //0x0140
UINT16 res12[7];
UINT32 pdi_config; //0x0150
UINT16 res13[86];
UINT16 ecat_interrupt_mask; //0x0200
UINT16 res14;
UALEVENTMASK al_event_mask; //0x0204
UINT16 res15[4];
UINT16 ecat_interrupt_request; //0x0210
UINT16 res16[7];
UALEVENT AlEvent; //0x0220
UINT16 res17[110];
UINT16 rx_error_counter[4]; //0x0300
UINT8 rx_error_cntforwarded[4]; //0x0308
UINT8 ecat_pu_errorcnt; //0x030c
UINT8 pdi_error_cnt; //0x030d
UINT16 res18;
UINT8 lost_link_cnt[4]; //0x0310
UINT16 res19[118];
UINT16 watchdog_divider; //0x0400
UINT16 res20[7];
UINT16 watchdog_time_pdi; //0x0410
UINT16 res21[7];
UINT16 watchdog_time_pd; //0x0420
UINT16 res22[15];
UINT16 watchdog_status_pd; //0x0440
UINT8 watchdog_cnt_pd; //0x0442
UINT8 watchdog_cnt_pdi; //0x0443
UINT16 res23[94];
TEEPROM_DEF eeprom_interface; //0x0500
TMII mii_man; //0x0510
UINT16 res24[117];
TFMMU fmmu_register[16]; //0x0600
UINT16 res25[128];

```

```

TSYNCFMAN sm_register[8];                                //0x0800
UINT16 res26[64];
TDC dc_register;                                         //0x0900
UINT16 res32[512];
UINT8 esc_specific_register;                            //0x0e00
UINT32 digital_io_outpd;                             //0x0f00
UINT16 res33[6];
UINT16 general_purp_outputs;                         //0x0f10
UINT16 res34[3];
UINT16 general_purp_inputs;                          //0x0f18
UINT16 res35[51];
UINT8 user_ram[128];                                  //0x0f80
}TESC_REG;

//-----  

// 全局变量定义  

//-----  

// ESC操作指针定义
TESC_REG MEMTYPE * pEsc;                               //ESC寄存器入口指针
// ESC寄存器缓存变量
UALEVENT EscAlEvent;
// 保存AL Status的值
UINT8 nAlStatus;
// 当状态变换出现错误时，保存最后的状态，如果状态机重新初始化，该变量保存
UINT8 nAlStatusFailed;
// AL StatusCode的值，该值应被写入AL-StatusCode寄存器(0x134)
UINT16 nAlStatusCode;
// 保存AL Control的值
UINT8 nAlControl;
// 保存SM2的输入长度，由应用层设定
UINT16 nPdInputSize;
// 保存SM3的输出长度，由应用层设定
UINT16 nPdOutputSize;
// 保存SM2的地址
UINT16 nEscAddrOutputData;
// 保存SM3的地址
UINT16 nEscAddrInputData;
// SM2输出数据指针
UINT8 MEMTYPE * pPdOutputData;
// SM3输入数据指针
UINT8 MEMTYPE * pPdInputData;

```

```

UINT16 u16SendMbxSize; // 非周期发送SM通道字节数
UINT16 u16ReceiveMbxSize; // 非周期接收SM通道字节数
UINT16 u16EscAddrReceiveMbx; // 非周期接收SM通道地址
UINT16 u16EscAddrSendMbx; // 非周期发送SM通道地址
UINT8 * pMbxWriteData; // SM0输出数据指针
UINT8 * pMbxReadData; // SM1输入数据指针

// 程序运行状态
UINT8 m_maxsyncman; // ESC支持的最大SM数目
BOOL m_mbxrunning; // 标志非周期性通信是否运行
BOOL m_pdooutrun; // 标志周期性输出是否运行
BOOL m_pdoinrun; // 标志周期性输入是否运行

// ESC中断使能标志 (SM2/3或SYNC0/1事件中断)
// 在StartInputHandler中设置，在StopInputHandler复位
BOOL bEscIntEnabled;

// 标志输入和输出是否运行在3个缓存区模式
BOOL b3BufferMode;

// 标志看门狗是否触发
BOOL bWdTrigger;

// 标志在OP状态，在StartInputHandler中设置，在StopInputHandler复位
BOOL bEcatOutputUpdateRunning;

// 通信数据存储
UINT8 aPdOutputData[MAX_PD_OUTPUT_SIZE]; // 周期性输出数据
UINT8 aPdInputData[MAX_PD_INPUT_SIZE]; // 周期性输入数据
UINT8 aMbOutputData[MAX_MB_OUTPUT_SIZE]; // 非周期性输出数据
UINT8 aMbInputData[MAX_MB_INPUT_SIZE]; // 非周期性输入数据
UINT32 mb_counter, pd_counter; // 通信计数

```

7.2 从站基本操作

(1) 对 ESC 控制寄存器的访问

在程序当中定义一个 TESC_REG 类型指针，使其指向 ESC 芯片的地址，则可以用这个指针来操作 ESC 内部寄存器。该操作的示例程序如下：

```

TESC_REG_MEMTYPE * pEsc;           //ESC寄存器入口指针
                                    // MEMTYPE为平台相关宏定义
pEsc = (TESC_REG_MEMTYPE *) ESC_REG_ENTRY; // ESC_REG_ENTRY为地址宏定义

```

例如，读状态机控制寄存器 al_ctl 到变量 alcontrol 的程序为：

```

UINT16 alcontrol;                // 变量定义
alcontrol = pEsc->al_ctl;        // 用pEsc读取应用层状态控制值

```

(2) 设置应用层状态

设置应用层状态的源程序代码为：

```

//-----//
// SetAlStatus(): 设置从站AL状态寄存器
//alstatus: 0x130~0x131
//alstatuscode: 0x134~0x135
//-----//
void SetAlStatus(UINT16 alstatus, UINT16 alstatuscode)
{
    pEsc->al_status = alstatus;          //为AL状态寄存器赋值
    if(alstatuscode != 0xFF)
        pEsc->al_statuscode = alstatuscode; //为AL状态代码寄存器赋值
}

```

(3) 设置事件中断屏蔽寄存器

设置事件中断屏蔽寄存器的源程序代码为：

```

//-----//
// set_intmask(): 设置从站中断屏蔽寄存器0x204~0x205
//intMask:中断屏蔽码
//-----//
void set_intmask(UINT16 intMask)
{
    UINT16 mask;

    mask = pEsc->al_event_mask.Word[0];
    mask = mask | intMask;
    pEsc->al_event_mask.Word[0] = mask;
}

//-----//
// reset_intmask(): 复位从站中断屏蔽码0x204~0x205
//intMask:      中断屏蔽码
//-----//
void reset_intmask(UINT16 intMask)
{
}

```

```

    UINT16 mask;

    mask = pEsc->al_event_mask.Word[0];
    mask = mask & intMask;
    pEsc->al_event_mask.Word[0] = mask;
}

```

(4) SM 通道操作

SM 通道操作的源程序为：

```

//-----
// enable_syncmchannel(): 使能SM运行
//channel: 通道号
//清除SM_PDI_CTL寄存器以Bit0使能SM
//-----
void enable_syncmchannel(UINT8 channel)          //使能SM
{
    pEsc->sm_register[channel].sm_register_pdctrl &=
        ~((UINT8) SM_PDIDISABLE);
}

//-----
// disable_syncmchannel(): 禁止SM运行
//channel: 通道号
//清除SM_PDI_CTL寄存器Bit0以使能SM
//-----
//brief 禁用SyncManager
//param channel:通道号
void disable_syncmchannel(UINT8 channel)
{
    pEsc->sm_register[channel].sm_register_pdctrl |=
        ((UINT8) SM_PDIDISABLE);
}

//-----
// get_sm(): 获取SyncManager寄存器
//channel: 通道号
//返回SM指针
//-----
TSYNCMAN * get_sm(UINT8 channel)
{
    return &(pEsc->sm_register[channel]);
}

```

7.3 从站驱动程序总体结构

EtherCAT 从站以 EtherCAT 从站控制器(ESC)芯片为核心, ESC 实现 EtherCAT 数据链路层, 完成数据的接收和发送以及错误处理。从站使用微处理器操作 ESC 芯片, 实现应用层协议, 包括以下任务:

- ① 微处理器初始化、通信变量和 ESC 寄存器初始化;
- ② 通信状态机处理, 完成通信初始化: 查询主站的状态控制寄存器, 读取相关配置寄存器, 启动或终止从站相关通信服务;
- ③ 周期性数据处理, 实现过程数据通信: 从站以查询模式(自由运行模式)或同步模式(中断模式)处理周期性数据和应用层任务。

图 7.1 为支持查询模式的从站驱动程序流程, 此模式下周期性数据在函数 free_run()中处理, 而同步模式下周期性数据在中断服务程序中处理。详细周期性数据处理流程见 7.4 节。

ESC 通信寄存器由主站配置, 从站程序只需要从中读取有效数据即可, 相关基本寄存器如表 7.1 所列。

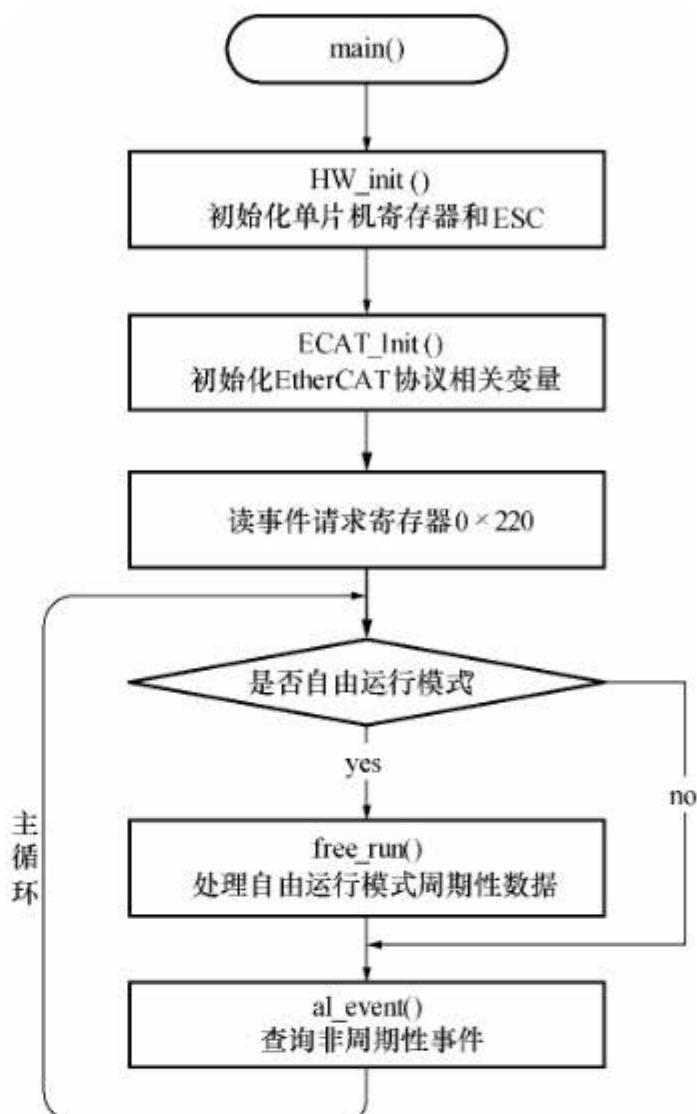


图 7.1 从站程序总体流程图

表 7.1 从站操作相关的基本寄存器

编 号	地址(长度)	名 称	读/写	操 作
1	0x120(2)	应用层状态控制	读	读取主站发出的状态改变指令
2	0x130(6)	应用层状态及状态码	写	返回从站实际状态及状态码
3	0x204(4)	应用事件中断屏蔽	写	设置事件触发中断信号
4	0x220(4)	应用事件请求	读	运行中查询发生的事件
5	0x800(32)	SM 配置数据	读	读取 SM 通道内存的起始地址和长度
6	0x098E	SYNC0 信号状态	读	读取此寄存器响应 SYNC0 中断

事件请求寄存器 0x0220~0x0223 和事件屏蔽寄存器 0x0204~0x0207 的定义按位对应, 如表 7.2 所列。使用函数 set_intmask() 写事件屏蔽寄存器, 设置相应事件触发中断信号。在中断程序中读取事件请求寄存器, 判断事件类型, 并作相应处理。

表 7.2 应用层事件请求寄存器定义

位	描述	复位方式
0	状态控制寄存器改变事件， 0：状态控制寄存器没有改变 1：状态控制寄存器被写操作	读状态控制寄存器 0x0120
1	锁存事件， 0：锁存输入没有变化 1：锁存输入至少变化一次	读锁存事件时间寄存器 0x9B0~0x9CF
2	SYNC0 引脚状态映射，当 R0x151.3=1 时有效	读 SYNC0 状态寄存器 0x098E
3	SYNC1 引脚状态映射，当 R0x151.7=1 时有效	读 SYNC1 状态寄存器 0x098F
4	SM 激活寄存器改变， 0：没有任何变化 1：至少一个 SM 激活状态改变	读取 SM 激活寄存器 0x980+i×6
5~7	保留	
8	SM _n 状态映射 (n=0~15)， 0：没有 SM 通道 n 事件 1：有 SM 通道 n 事件	读取 SM 通道所管理的存储器空间，地址在 SM 配置寄存器中
...		
23		

从站驱动程序主函数的源程序代码如下：

```

//-----  

// main(): 程序主函数  

// 执行初始化过程，并进入主循环  

//-----  

void main(void)  

{  

    HW_init(); // 初始化微处理器寄存器  

    ECAT_init(); // 初始化通信变量和ESC寄存器  

    while (1) // 主循环  

    {  

        // 读应用层事件请求寄存器，EscAlEvent为全局变量，在头文件中定义  

        EscAlEvent = pEsc->AlEvent;  

        if (!bEscIntEnabled)  

        { // 未使能中断，处于自由运行模式  

            free_run(); // 自由运行模式，查询周期性数据，见7.4.2  

        }  

        al_event(); // 应用层事件处理，包括状态机和非周期性通信等，见7.5  

    };  

}

```

HW_init()函数用以初始化微处理器Atmega128，并使能外部中断信号引脚1响应，其源程序代码如下：

```
-----  
// HW_init(): 初始化通信变量和ESC寄存器  
-----  
void HW_init()  
{  
    MCUCR = 0xC0;      // 使能外部存储空间访问功能  
    EICRA = 0xFF;      // 配置中断信号为上升沿触发  
    EICRB = 0xff;      // 配置中断信号为上升沿触发  
    EIMSK = 0x20;      // 使能外部中断1响应  
}
```

ECAT_init()函数用以初始化通信控制变量和ESC寄存器，其程序源代码如下：

```
-----  
// ECAT_init(): 初始化通信变量和ESC寄存器  
-----  
void ECAT_init()  
{  
    // 给指向ESC的指针变量赋值  
    pEsc = (TESC_REG_MEMTYPE *) ESC_REG_ENTRY;  
    // 清除事件屏蔽寄存器0x204~0x205  
    pEsc->al_event_mask.Word[0] = 0;  
    // 清除事件请求寄存器0x206~0x207  
    pEsc->al_event_mask.Word[1] = 0;  
    m_maxsyncman = 0;  
    // 读取ESC所支持的SM通道数目  
    m_maxsyncman = pEsc->sm_supported;  
    nAIStatus = STATE_INIT;  
    // 设置当前状态为“初始化状态”  
    SetAIStatus(nAIStatus, 0);  
    // 初始化通信变量  
    nPdInputSize = 0;  
    nPdOutputSize = 0;  
    bEcatLocalError = 0;  
    bEscIntEnabled = FALSE;  
}
```

7.4 从站周期性数据的处理

从站设备可以运行在自由运行模式和同步模式，自由运行模式使用查询方式处理周期性数据，同步模式则在中断服务例程中处理周期性过程数据。程序中使用全局变量 bEscIntEnabled 控制运行模式：bEscIntEnabled 等于 0 时，使用自由运行模式；bEscIntEnabled 等于 1 时，使用同步模式。在初始化阶段根据主站对 SM 的配置来初始化变量 bEscIntEnabled，以决定当前的运行模式。

7.4.1 同步运行模式

使用同步运行模式时，在中断服务例程中处理周期性过程数据。由于其他应用事件也可能触发中断，所以需要在中断服务例程中判断中断源以做相应处理。从站使用 Atmega128 的外部中断引脚 1 作为中断输入，并在 HW_init() 函数中对其进行初始化。图 7.2 为中断服务例程流程图，其执行过程如下：

- ① 读取事件请求寄存器 0x220~0x223 到变量 EscAlEvent；
- ② 从站程序查询 SM2 事件，如果发生 SM2 事件，则从 ESC 的 SM2 管理的存储区读取周期性输出数据；
- ③ 如果输出有效，则将输出数据映射到相应的输出变量，并执行硬件输出操作；在 Safe_Op 状态下，虽然有输出数据，但是从站不执行硬件输出操作；程序中使用全局变量 bEcatOutputUpdateRunning 表示输出是否有效；
- ④ 读取硬件输入操作，将输入数据写入 SM3 管理的存储区，等待下个数据帧读取。

中断服务例程的程序源代码如下：

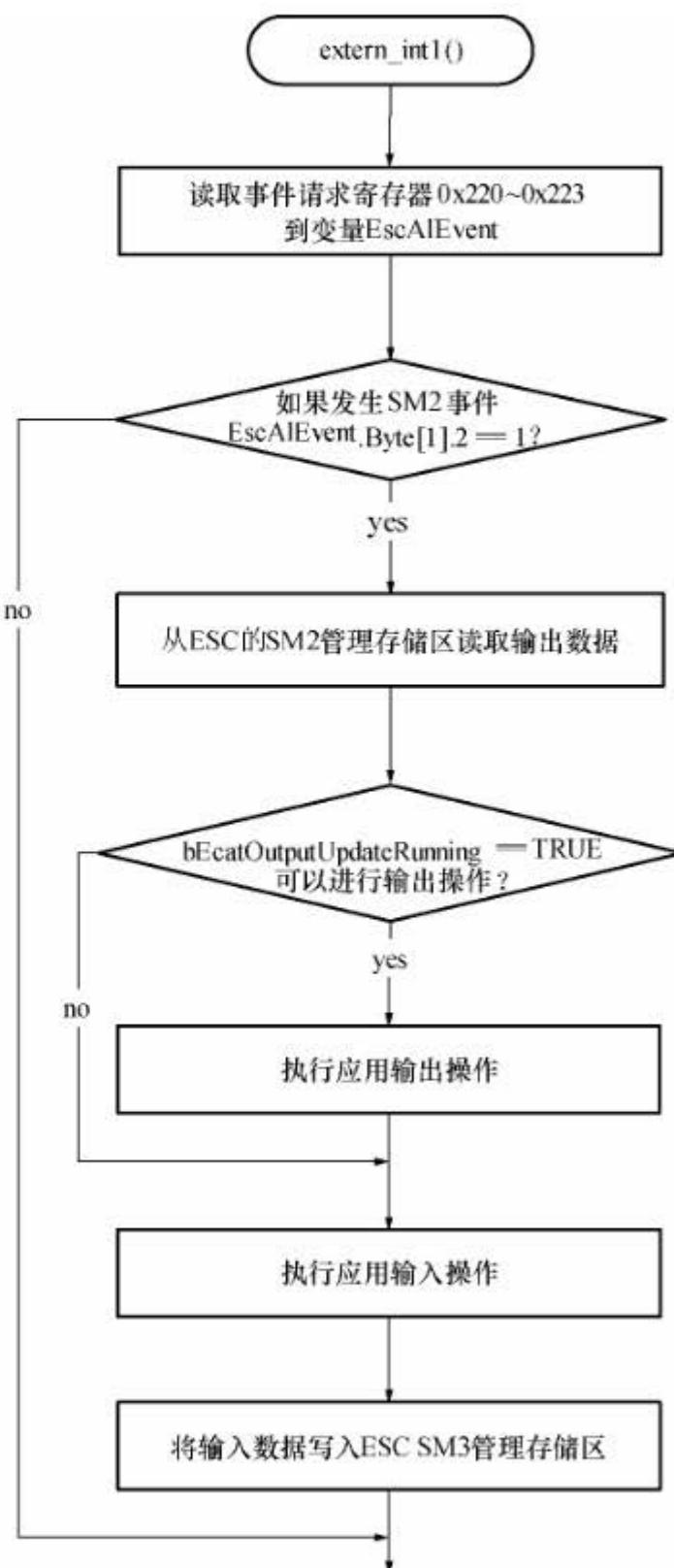


图 7.2 同步运行模式中断服务例程流程图

```

//-----
// exter_int1(): 中断服务例程, 处理周期性数据通信
//-----
interrupt [EXT_INT1] void exter_int1(void)
{
    // 读应用层事件请求寄存器
    EscAlEvent = pEsc->AlEvent;
    if((EscAlEvent.Byte[1]) & (PROCESS_OUTPUT_EVENT >> 8))
    {
        // 有SM2事件发生
        memcpy(aPdOutputData, pPdOutputData, nPdOutputSize);
        // 从ESC SM2管理的存储区读取周期性输出数据
        // aPdOutputData: ec_def.h中定义的数组, 保存输出数据
        // pPdOutputData: ec_def.h中定义的指针, 指向SM2管理的内存区
        // nPdOutputSize: ec_def.h中定义的变量, 表示SM2管理内存区的容量
        if(bEcatOutputUpdateRunning == TRUE) //该变量的定义见7.1
            memcpy(aPdInputData, aPdOutputData, min(nPdOutputSize,
                nPdInputSize));
        //如果输出有效, 执行输出操作, 本例中直接将输出数据映射到输入数据
        //aPdInputData: ec_def.h中定义的数组, 保存输入数据
        //nPdInputSize: ec_def.h中定义的变量, 表示SM3管理内存区的容量
        memcpy(pPdInputData, aPdInputData, nPdInputSize);
        // 执行输入操作, 将输入数据写入SM3管理的存储区
    }
}

```

7.4.2 自由运行模式

自由运行模式下, 从站程序不使用中断, 而是在主函数中查询输出事件, 并执行相关操作。其详细流程如图 7.3 所示。

自由运行模式的程序源代码如下:

```

//-----
// free_run(): 自由运行模式, 查询周期性输出数据
//-----
void free_run()
{
    if((EscAlEvent.Byte[1]) & (PROCESS_OUTPUT_EVENT >> 8))
    {
        // 有SM2事件发生
        memcpy(aPdOutputData, pPdOutputData, nPdOutputSize);
        // 从ESC SM2管理的存储区读取周期性输出数据
        // aPdOutputData: ec_def.h中定义的数组, 保存输出数据
        // pPdOutputData: ec_def.h中定义的指针, 指向SM2管理的内存区
    }
}

```

```

//nPdOutputSize: ec_def.h中定义的变量, 表示SM2管理内存区的容量
if (bEcatOutputUpdateRunning == TRUE) //定义见7.1
    memcpy(aPdInputData, aPdOutputData,
           min (nPdOutputSize, nPdInputSize));
// 如果输出有效, 则执行输出操作, 本例中直接将输出数据映射到输入数据
//aPdInputData: ec_def.h中定义的数组, 保存输入数据
//nPdInputSize: ec_def.h中定义的变量, 表示SM3管理内存区容量
memcpy (pPdInputData, aPdInputData, nPdInputSize);
// 执行输入操作, 将输入数据写入SM3管理的存储区
}
}

```

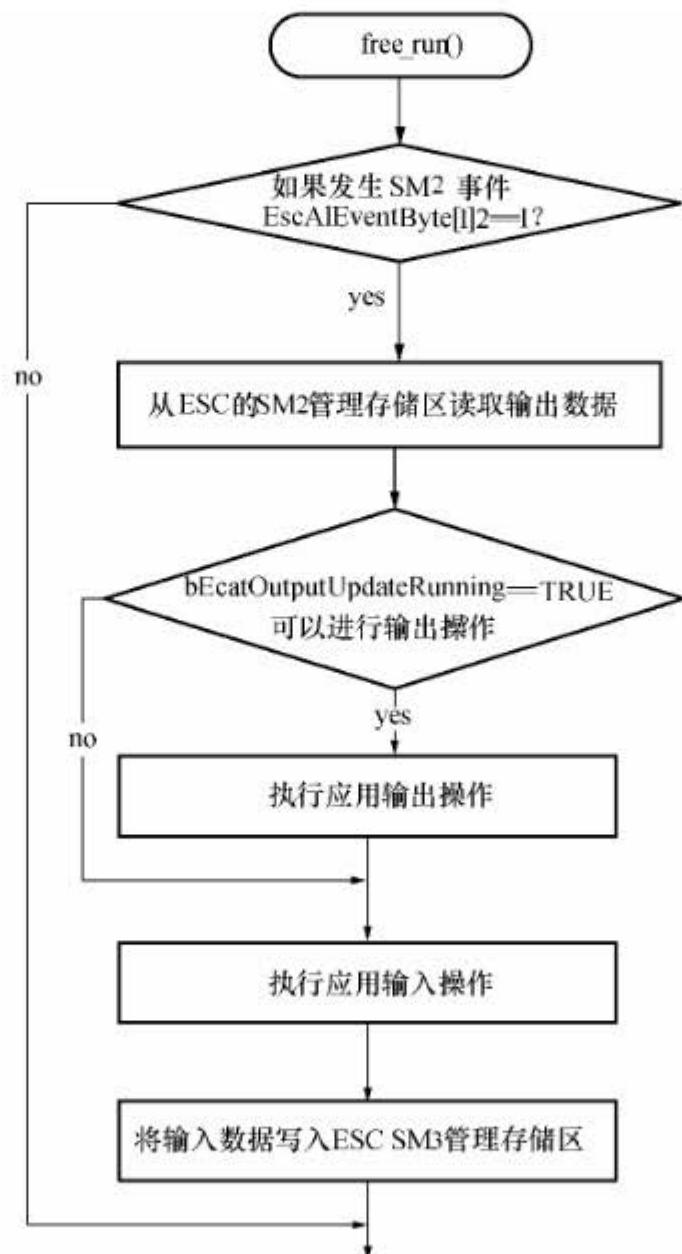


图 7.3 自由运行模式流程图

7.5 从站非周期性事件的处理

从站非周期性事件主要有状态改变事件和邮箱通信事件。在程序主函数的主循环中读取应用层事件请求寄存器以查询事件的发生。本程序只给出邮箱通信的接口，并未实现具体的应用层协议。在 7.6 节将详细介绍状态机处理。处理从站非周期性事件的程序源代码如下：

```

//-----
// al_event(): 查询处理应用层事件
// 读ESC中断事件请求寄存器，根据其中的有效位做相应的处理
//-----
void al_event(void)
{
    UINT16 alcontrol;

    // 读ESC中断事件请求寄存器
    EscAlEvent = pEsc->AlEvent;

    // 判断是否AL控制变化事件发生
    if (EscAlEvent.Byte[0] & AL_CONTROL_EVENT)
    {
        // 是，则读AL控制寄存器0x120以响应事件
        alcontrol = pEsc->al_ctl;
        nAlControl = alcontrol;
        // 调用状态机处理函数
        al_statemachine(alcontrol); // 见7.6.1节
    }
    // 如果非周期性通信在运行
    if (m_mbrunning) // 全局变量，其定义见7.1
    {
        // 判断是否有非周期性输出数据（SM2通道事件）到达
        if ((EscAlEvent.Byte[1]) & (MAILBOX_WRITE_EVENT >> 8))
        {
            mb_process(); // 处理邮箱通信的接口
        }
    }
}

```

7.6 从站状态机的处理

从站在主函数的主循环中查询状态机改变事件请求位，如果发生变化，则执行状态机管理机制。状态机管理流程如图 7.4 所示。从站程序首先检查当前状态转化必须的 SM 配置是

否正确，如果正确，则根据转化要求开始相应的通信数据处理。从站从高级别状态向低级别状态变化时，则停止相应的通信数据处理。

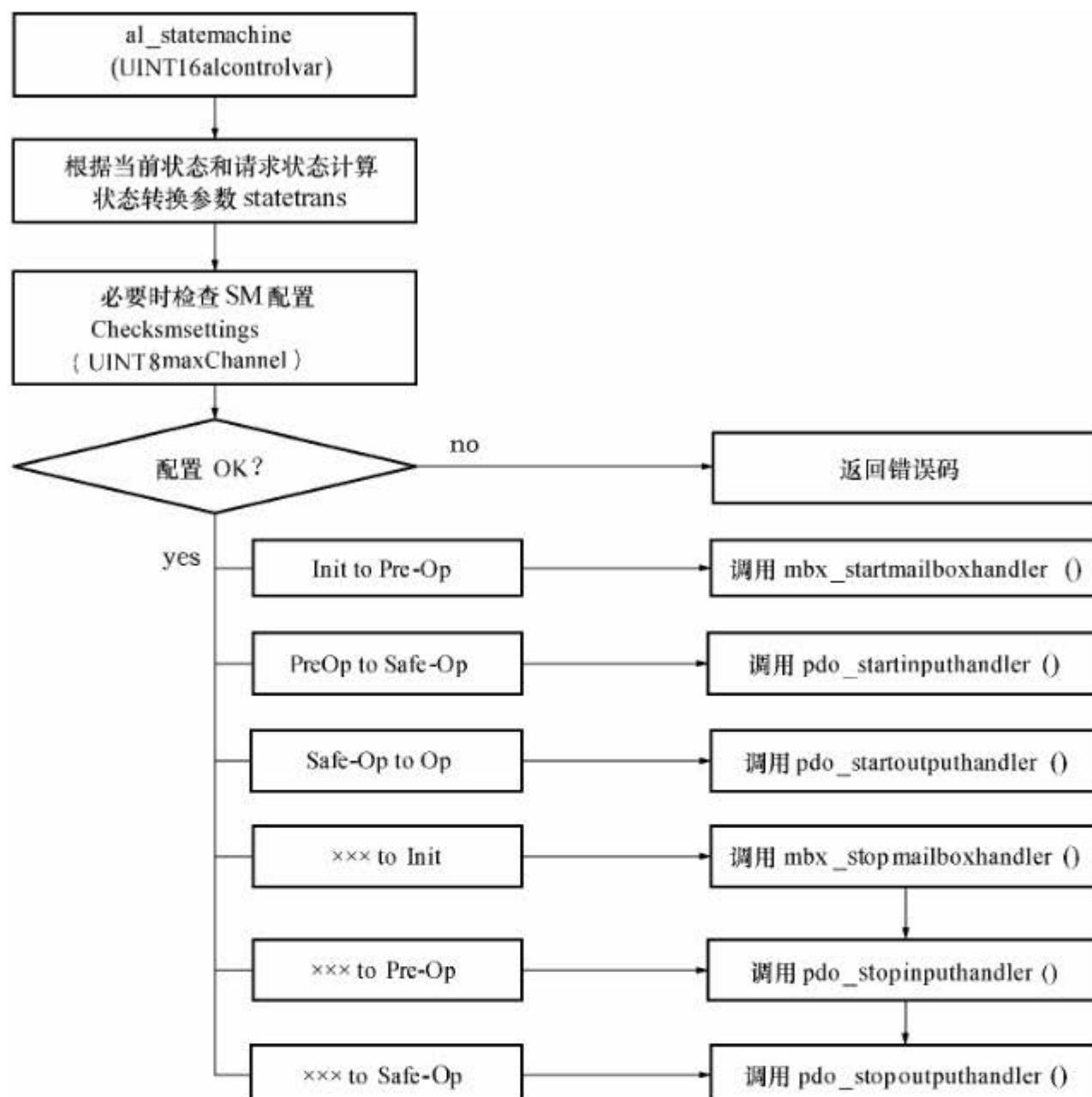


图 7.4 EtherCAT 从站状态机处理流程

7.6.1 状态机处理流程

al_statemachine()函数实现了图7.4所示的状态机处理流程，其程序源代码如下：

```

//-----  

// al_statemachine (): 处理AL状态机  

// alcontrolvar: 当前状态  

//-----  

void al_statemachine (UINT16 alcontrolvar)  

{
    UINT8 result = 0;  

    UINT8 statetrans;  

    UINT8 val;
  
```

```

    UINT8 al;
    al = alcontrolvar;

    if(alcontrolvar & STATE_ERRACK)
    {
        nAlStatus &= ~STATE_ERROR;
    }
    else if((nAlStatus & STATE_ERROR) && (((UINT8)alcontrolvar
    & STATE_MASK) > (nAlStatus & STATE_MASK)))
        return;

    alcontrolvar &= STATE_MASK;
    statetrans = nAlStatus;           //得到目前的状态
    statetrans <<= 4;
    statetrans += alcontrolvar;     //得到转换状态变量

    // 检查SM设置
    switch(statetrans)
    {
        case INIT_2_PREOP:
        case OP_2_PREOP:
        case SAFEOP_2_PREOP:
        case PREOP_2_PREOP:
            val = MAILBOX_READ + 1;
            result = checksmsettings(val);           // 见7.6.2节
            //在此状态检查SM0/SM1的设置
            break;
        case PREOP_2_SAFEOP:
        case SAFEOP_2_OP:
        case OP_2_SAFEOP:
        case SAFEOP_2_SAFEOP:
        case OP_2_OP:
            result = checksmsettings(m_maxsyncman); // 见7.6.2节
            //在此检查所有的SM
            break;
    }

    // 如果SM设置正确，则进行下一步处理
    if(result == 0)
    {
        switch(statetrans)

```

```

{
    case INIT_2_PREOP:
        result = mbx_startmailboxhandler(); // 见7.6.3节
        break;
    case PREOP_2_SAFEOP:
        result = pdo_startinpuhandler(); // 见7.6.4节
        // 设置过程数据中断，使能SM2, SM3
        break;
    case SAFEOP_2_OP:
        result = pdo_startoutputhandler(); // 见7.6.5节
        break;
    case OP_2_INIT:
    case SAFEOP_2_INIT:
    case PREOP_2_INIT:
        mbx_stopmailboxhandler(); // 停止邮箱应用程序，见7.6.6节
    case OP_2_PREOP:
    case SAFEOP_2_PREOP:
        result = pdo_stopinpuhandler(); // 见7.6.6节
        if(result != 0)
            break;
    case OP_2_SAFEOP:
        result = pdo_stopoutputhandler(); // 见7.6.6节
        break;

    case INIT_2_INIT:
    case PREOP_2_PREOP:
    case SAFEOP_2_SAFEOP:
    case OP_2_OP:
        result = NOERROR_NOSTATECHANGE;
        break;

    case INIT_2_SAFEOP:
    case INIT_2_OP:
    case PREOP_2_OP:
        result = ALSTATUSCODE_INVALIDALCONTROL;
        break;
    default:
        // setalstatus(0x01, 0x00);
        result = ALSTATUSCODE_UNKNOWNALCONTROL;
        break;
}

```

```

    }

}

else // 如果SM设置不正确，则进行错误处理
{
    switch (nAlStatus)
    {
        case STATE_OP:
            pdo_stopoutputhandler(); // 见7.6.6节
            break;
        case STATE_SAFEOP:
            pdo_stopinputhandler(); // 见7.6.6节

        case STATE_PREOP:
            if(result == ALSTATUSCODE_INVALIDMBXCFGINPRE)
            {
                mbx_stopmailboxhandler(); // 见7.6.6节
                nAlStatus = STATE_INIT;
            }
            else
            {
                nAlStatus = STATE_PREOP;
            }
            break;
    }
}

// 设置alStatus和alStatusCode
if((UINT8)alcontrolvar != (nAlStatus & STATE_MASK))
{
    if(result != 0)
    {
        nAlStatusFailed = nAlStatus;
        nAlStatus |= STATE_CHANGE; // 没有成功，状态机状态不变
    }
    else
    {
        if(nAlStatusCode != 0)
        {
            result = nAlStatusCode;
            nAlStatusFailed = alcontrolvar;
            alcontrolvar |= STATE_CHANGE;
        }
    }
}

```

```
        }

        else if(alcontrolvar <= nAlStatusFailed)

        {

            result = 0xFF;

        }

        else

            nAlStatusFailed = 0;

            nAlStatus = alcontrolvar; //为Alstatus 寄存器赋值

        }

        SetAlStatus(nAlStatus, result); // 该函数在7.2中介绍

        nAlStatusCode = 0;

    }

    else

    {

        SetAlStatus(nAlStatus, 0xFF); // 该函数在7.2中介绍

    }

}
```

7.6.2 检查 SM 通道设置

在进入“Pre-Op”状态之前时，需要读取并检查邮箱通信相关SM通道0和1的配置，进入“Safe-Op”之前需要检查周期性过程数据通信使用的SM2和SM3的设置。需要检查的SM通道的设置内容有：

- ① SM通道的大小；
 - ② SM通道的设置是否重叠，特别注意三个缓存区时应该预留配置长度3倍大小的空间；
 - ③ SM通道起始地址应该为偶数；
 - ④ SM通道应该被使能。

检查SM通道设置的程序源代码如下。

```
//-----  
// checksmsettings (): 检查SyncManager设置  
// maxChannel:最大通道数  
//返回执行结果  
//-----  
UINT8 checksmsettings(UINT8 maxChannel)  
{  
    UINT8 i;  
    UINT8 result = 0;  
    UINT8 smFailed = 0;  
    TSYNCFMAN MEMTYPE *pSyncMan;  
  
    //检查接收Mailbox(SyncManager通道)的SyncManager参数
```

```

pSyncMan = get_sm(MAILBOX_WRITE);

if ( (pSyncMan->sm_register_activate & SM_ECATENABLE)
    != SM_ECATENABLE )
    //接收Mailbox没有使能
    result = ALSTATUSCODE_INVALIDMBXCFGINPRE;
else if ( (pSyncMan->sm_register_control & SM_INITMASK)
    != (ONE_BUFFER | SM_WRITESETTINGS) )
    //接收Mailbox不可被主站写或SM不是一个缓存区的模式
    result = ALSTATUSCODE_INVALIDMBXCFGINPRE;
else if ( pSyncMan->sm_length < MIN_MBX_SIZE )
    //接收Mailbox长度过小
    result = ALSTATUSCODE_INVALIDMBXCFGINPRE;
else if ( pSyncMan->sm_length > MAX_MBX_SIZE )
    //接收Mailbox长度过大
    result = ALSTATUSCODE_INVALIDMBXCFGINPRE;
else if ( pSyncMan->sm_physical_addr < MIN_MBX_WRITE_ADDRESS )
    //接收Mailbox地址过小
    result = ALSTATUSCODE_INVALIDMBXCFGINPRE;
else if ( pSyncMan->sm_physical_addr > MAX_MBX_WRITE_ADDRESS )
    //接收Mailbox地址过大
    result = ALSTATUSCODE_INVALIDMBXCFGINPRE;
else if ( (pSyncMan->sm_physical_addr & 0x0001) != 0 )
    //接收Mailbox地址不是偶数
    result = ALSTATUSCODE_INVALIDMBXCFGINPRE;

//如果SyncManager长度为零，则返回
if(pSyncMan->sm_length == 0)
    result = ALSTATUSCODE_NOERROR;

if ( result == 0 )
{
    //检查发送Mailbox(SyncManager通道)的SyncManager参数
    pSyncMan = get_sm(MAILBOX_READ);
    if ( (pSyncMan->sm_register_activate & SM_ECATENABLE)
        != SM_ECATENABLE )
        //发送Mailbox没有使能
        result = ALSTATUSCODE_INVALIDMBXCFGINPRE;
    else if ( (pSyncMan->sm_register_control & SM_INITMASK) !=
        (ONE_BUFFER | SM_READSETTINGS) )
        //发送Mailbox不可被主站写或SM不是一个缓存区的模式
        result = ALSTATUSCODE_INVALIDMBXCFGINPRE;
}

```

```

else if ( pSyncMan->sm_length < MIN_MBX_SIZE )
    //发送Mailbox长度过小
    result = ALSTATUSCODE_INVALIDMBXCFGINPRE;
else if ( pSyncMan->sm_length > MAX_MBX_SIZE )
    //发送Mailbox长度过大
    result = ALSTATUSCODE_INVALIDMBXCFGINPRE;
else if ( pSyncMan->sm_physical_addr
< MIN_MBX_READ_ADDRESS )
    //发送Mailbox地址过小
    result = ALSTATUSCODE_INVALIDMBXCFGINPRE;
else if ( pSyncMan->sm_physical_addr
> MAX_MBX_READ_ADDRESS )
    //发送Mailbox地址过大
    result = ALSTATUSCODE_INVALIDMBXCFGINPRE;
else if ( (pSyncMan->sm_physical_addr & 0x0001) != 0 )
    //发送Mailbox地址不是偶数
    result = ALSTATUSCODE_INVALIDMBXCFGINPRE;

//如果SyncManager长度为零，则返回
if (pSyncMan->sm_length == 0)
    result = ALSTATUSCODE_NOERROR;

if (result != 0)
    smFailed = MAILBOX_READ;
}

else
    smFailed = MAILBOX_WRITE;

if ( result == 0 && maxChannel > PROCESS_DATA_IN )
{
    b3BufferMode = TRUE;
    //检查输入(SyncManager通道)的SyncManager参数
    pSyncMan = get_sm(PROCESS_DATA_IN);

    if ( (pSyncMan->sm_register_activate & SM_ECATENABLE)
        != 0 && pSyncMan->sm_length == 0 )
        //SyncManager3长度为0且SyncManager3没有激活
        result = SYNCMANCHSETTINGS+1;
    else if ( (pSyncMan->sm_register_activate & SM_ECATENABLE)
        == SM_ECATENABLE )
    {

```

```

//大小匹配

if ( (pSyncMan->sm_register_control & SM_PDINITMASK)
    == SM_READSETTINGS )
{
    //设置匹配

    if ( ( (nAIStatus == STATE_PREOP)
        && (pSyncMan->sm_physical_addr >=
            MIN_PD_READ_ADDRESS) && (pSyncMan->sm_physical_addr
            <= MAX_PD_READ_ADDRESS) )
        || ( (nAIStatus != STATE_PREOP)
            && (pSyncMan->sm_physical_addr ==
                nEscAddrInputData) ) )
    {

        //地址匹配

        if ( (pSyncMan->sm_register_control &
            SM_INITMASK) == (ONE_BUFFER|SM_READSETTINGS) )
            //输入为1个缓存的模式，复位标识b3BufferMode
            b3BufferMode = FALSE;

    }
    else
        //输入地址超出允许范围或者在SAFEOP或OP状态被改变
        result = SYNCMANCHADDRESS+1;
}
else
    //输入设置不匹配
    result = SYNCMANCHSETTINGS+1;
}

else if ( pSyncMan->sm_length != 0 || nPdInputSize != 0 )
    //SM3输入大小不为零而SM3没有激活
    result = SYNCMANCHSIZE+1;

//如果长度为零，则返回

if (pSyncMan->sm_length == 0)
{
    nPdInputSize = 0x0;
    nPdOutputSize = 0x0;
    result = ALSTATUSCODE_NOERROR;
}

if ( result != 0 )
{

```

```

        result = ALSTATUSCODE_INVALIDSMINCFG;
        smFailed = PROCESS_DATA_IN;
    }
}

if ( result == 0 && maxChannel > PROCESS_DATA_OUT )
{
    //检查输出(SyncManager通道)的SyncManager参数
    pSyncMan = get_sm(PROCESS_DATA_OUT);
    nPdOutputSize = pSyncMan->sm_length;
    if ( (pSyncMan->sm_register_activate & SM_ECATENABLE)
        != 0 && pSyncMan->sm_length == 0 )
        //SM2大小为零而SM2没有激活
        result = SYNCMANCHSETTINGS+1;
    else if ( (pSyncMan->sm_register_activate & SM_ECATENABLE)
        == SM_ECATENABLE )
    {
        //SyncManager通道激活，输出大小必须大于0
        if(nPdOutputSize != 0)
        {
            //大小匹配
            if ((pSyncMan->sm_register_control & M_PDINITMASK)
                == SM_WRITESETTINGS )
            {
                //设置匹配
                if ( ( nAlStatus == STATE_PREOP )
                    && ( pSyncMan->sm_physical_addr >=
                        MIN_PD_WRITE_ADDRESS ) &&
                        ( pSyncMan->sm_physical_addr <=
                            MAX_PD_WRITE_ADDRESS ) )
                    || ( ( nAlStatus != STATE_PREOP )
                        && ( pSyncMan->sm_physical_addr
                            == nEscAddrOutputData ) ) )
                {
                    //地址匹配
                    {
                        //检查看门狗触发器是否使能
                        if (pSyncMan->sm_register_control
                            & WATCHDOG_TRIGGER)
                            bWdTrigger = TRUE;
                        else

```

```

        bWdTrigger = FALSE;
        if ( (pSyncMan->sm_register_control
          & SM_INITMASK) == (ONE_BUFFER
          | SM_WRITESETTINGS) )
          // 输出运行在1个缓存区模式
          // 复位标识b3BufferMode
          b3BufferMode = FALSE;
      }
    }
  else
    //输出地址在允许范围以外或在SAFEOP (或OP状态) 被修改
    result = SYNCMANCHADDRESS+1;
}
else
  //输出设置不匹配
  result = SYNCMANCHSETTINGS+1;
}
else
  //输出大小不匹配
  result = SYNCMANCHSIZE+1;
}
else if ( pSyncMan->sm_length != 0 || nPdOutputSize != 0 )
  //输出大小不为零而SM2通道没有激活
  result = SYNCMANCHSIZE+1;

//SyncManager长度为零, 返回
if(pSyncMan->sm_length == 0)
  result = ALSTATUSCODE_NOERROR;

if ( result != 0 )
{
  result = ALSTATUSCODE_INVALIDSMOUTCFG;
  smFailed = PROCESS_DATA_OUT;
}

if ( result == 0 )
{
  //读剩余的SM通道的使能字节, 以应答SM变换的中断
  for (i = maxChannel; i < m_maxsyncman; i++)
  {

```

```

        pSyncMan = get_sm(i);

    }

    return result;
}

```

7.6.3 启动邮箱数据通信

在设置从站为“Pre-Op”状态之前，如果邮箱通信SM配置正确，则启动邮箱通信处理，其程序源代码如下：

```

-----  

// mbx_startmailboxhandler(): 启动非周期性数据通信  

// 返回执行结果  

-----  

UINT8 mbx_startmailboxhandler(void)  

{  

    //获取Mailbox接收SyncManger  

    TSYNCMAN MEMTYPE * pSyncMan;  

    pSyncMan = get_sm(MAILBOX_WRITE);  

    //保存接收Mailbox数据长度  

    u16ReceiveMbxSize = pSyncMan->sm_length;  

    //保存接收Mailbox地址  

    u16EscAddrReceiveMbx = pSyncMan->sm_physical_addr;  

    pMbxWriteData = (UINT8 MEMTYPE *)  

        (ESC_REG_ENTRY + u16EscAddrReceiveMbx);  

    //获取Mailbox发送SyncManager  

    pSyncMan = get_sm(MAILBOX_READ);  

    //保存发送Mailbox数据长度  

    u16SendMbxSize = pSyncMan->sm_length;  

    //保存发送Mailbox地址  

    u16EscAddrSendMbx = pSyncMan->sm_physical_addr;  

    pMbxReadData = (UINT8 MEMTYPE *)  

        (ESC_REG_ENTRY + u16EscAddrSendMbx);  

    //检查Mailbox是否有内存重叠设置  

    if ((u16EscAddrReceiveMbx+u16ReceiveMbxSize) >  

        u16EscAddrSendMbx && (u16EscAddrReceiveMbx <  

        (u16EscAddrSendMbx+u16SendMbxSize)))

```

```

{
    return ALSTATUSCODE_INVALIDMBXCFGINPRE; //返回错误码
}

//使能Mailbox接收SyncManager通道
enable_syncmchannel(MAILBOX_WRITE);
//使能Mailbox发送SyncManager通道
enable_syncmchannel(MAILBOX_READ);
//表明Mailbox已经开始运行
m_mbxrunning = TRUE;

return 0;
}

```

7.6.4 启动周期性输入数据通信

在进入“Safe-Op”状态时，如果过程数据SM通道设置正确，则使能输入数据通道SM3，设置中断屏蔽寄存器，使输出数据SM2触发中断；如果支持分布时钟，则使SYNC0事件触发中断，启动周期性输入数据通信，此时输出数据通道SM2并不使能，输出数据无效。

```

-----+
// pdo_startinpuhandler(): 启动周期性输入数据通信
// 返回执行结果
-----+
UINT8 pdo_startinpuhandler(void)
{
    UINT16 nPdInputBuffer = 3;
    UINT16 nPdOutputBuffer = 3;
    TSYNCMAN MEMTYPE * pSyncMan;
    UINT16 intMask = 0; //用于设置AL-Event-Mask寄存器
    UINT8 dcControl;
    UINT32 cycleTime;

    pSyncMan = get_sm(PROCESS_DATA_OUT);
    nEscAddrOutputData = pSyncMan->sm_physical_addr;
    nPdOutputSize = pSyncMan->sm_length;
    pPdOutputData = (UINT8 MEMTYPE *)
        (ESC_REG_ENTRY + pSyncMan->sm_physical_addr);
    if(pSyncMan->sm_register_control & ONE_BUFFER)
        nPdOutputBuffer = 1;

    pSyncMan = get_sm(PROCESS_DATA_IN);
}

```

```

nEscAddrInputData = pSyncMan->sm_physical_addr;
nPdInputSize = pSyncMan->sm_length;
pPdInputData = (UINT8 MEMTYPE *)
    (ESC_REG_ENTRY + pSyncMan->sm_physical_addr);
if (pSyncMan->sm_register_control & ONE_BUFFER)
    nPdInputBuffer = 1;

//如果长度为零，则返回
if (pSyncMan->sm_length == 0)
    return ALSTATUSCODE_NOERROR;

if ( ((nEscAddrInputData+nPdInputSize*nPdInputBuffer) >
    u16EscAddrSendMbx && (nEscAddrInputData <
    (u16EscAddrSendMbx+u16SendMbxSize)))
    || ((nEscAddrInputData+nPdInputSize*nPdInputBuffer) >
    u16EscAddrReceiveMbx && (nEscAddrInputData <
    (u16EscAddrReceiveMbx+u16ReceiveMbxSize))))
{
    //SyncManager通道内存区域(输入)与Mailbox内存区域重叠
    return ALSTATUSCODE_INVALIDDSMINCFG;
}

if ( ((nEscAddrOutputData+nPdOutputSize*nPdOutputBuffer) >
    u16EscAddrSendMbx && (nEscAddrOutputData <
    (u16EscAddrSendMbx+u16SendMbxSize)))
    || ((nEscAddrOutputData+nPdOutputSize*nPdOutputBuffer) >
    u16EscAddrReceiveMbx && (nEscAddrOutputData <
    (u16EscAddrReceiveMbx+u16ReceiveMbxSize)))
    || ((nEscAddrOutputData+nPdOutputSize*nPdOutputBuffer) >
    nEscAddrInputData && (nEscAddrOutputData <
    (nEscAddrInputData+nPdInputSize))))
{
    // SyncManager通道内存区域(输出)与Mailbox内存
    // 或SyncManager通道内存区域重叠
    return ALSTATUSCODE_INVALIDDSMOUTCFG;
}

dcControl = pEsc->dc_register.activation;
if ( dcControl & (DC_SYNC0_ACTIVE | DC_SYNC1_ACTIVE) )
{
    //分布式时钟启用，检查SYNC0/SYNC1设置
}

```

```

    if ( dcControl != (DC_CYCLIC_ACTIVE | DC_SYNC_ACTIVE) )
        return ALSTATUSCODE_DCINVALIDSYNCCFG;

    //激活DC事件
    intMask = DC_EVENT_MASK;
    //表明从站运行在分布式时钟模式
    bDcSyncActive = TRUE;
    cycleTime = pEsc->dc_register.sync0_cyclic_time;
}

if ( nPdOutputSize != 0 )
{
    //激活SM通道2(输出)中断事件
    intMask |= PROCESS_OUTPUT_EVENT;
}
else
{
    // 激活SM通道3(输入)中断事件
    intMask |= PROCESS_INPUT_EVENT;
}

if ( nPdInputSize > 0 )
{
    //使能SyncManger通道
    enable_syncmchannel(PROCESS_DATA_IN);
    m_pdoinrun = TRUE;
}
if ( nPdOutputSize > 0 )
{
    if ( !bEcatLocalError )
        //如果没有错误，使能SyncManger通道
        enable_syncmchannel(PROCESS_DATA_OUT);
    m_pdooutrun = TRUE;
}
set_intmask( intMask );
return 0;
}

```

7.6.5 启动周期性输出数据通信

进入“Op”状态时，使能输出数据SM2，启动周期性输出数据通信，其程序的源代码

如下：

```

//-----  

// pdo_startoutputhandler (): 启动周期性输出通信数据处理  

//返回执行结果  

//-----  

UINT8 pdo_startoutputhandler(void)  

{  

    UINT16 result = 0;  

    if (nPdOutputSize > 0)  

    {  

        if ( bEcatLocalError && (result == 0 || NOERROR_INWORK) )  

        {  

            //没有错误，使能SyncManager2通道  

            enable_syncmanchannel(PROCESS_DATA_OUT);  

            bEcatLocalError = FALSE;  

        }  

        if (result != 0)  

        {  

            if ( result != NOERROR_INWORK )  

                bEcatLocalError = TRUE;  

            return result;  

        }  

        m_pdooutrun = TRUE;  

    }  

    //表明输出已经运行  

    bEcatOutputUpdateRunning = TRUE;  

    return 0;  

}

```

7.6.6 停止 EtherCAT 数据通信

在EtherCAT通信状态回退时停止相应的数据通信SM通道，其回退有3种方式：

- ① 从高状态退回“Safe-Op”时，停止周期性过程数据输出处理；
- ② 从高状态退回“Pre-Op”时，停止所有周期性过程数据处理；
- ③ 从高状态退回“Init”时，停止所有应用层数据处理。

停止应用数据处理的程序源代码如下：

```

//-----  

// mbx_stopmailboxhandler(): 停止非周期性数据通信  

//返回执行结果  

//-----

```

```

//-----
void mbx_stopmailboxhandler(void)
{
    //表明Mailbox已经停止
    m_mbxbusy = FALSE;
    //禁用Mailbox接收SyncManager通道
    disable_syncmchannel(MAILBOX_WRITE);
    //禁用Mailbox发送SyncManager通道
    disable_syncmchannel(MAILBOX_READ);
}

//-----
// pdo_stopinputhandler(): 停止周期性输入通信数据处理
// 返回执行结果
//-----
UINT8 pdo_stopinputhandler(void)
{
    //禁用SyncManger通道
    disable_syncmchannel(PROCESS_DATA_OUT);
    //复位AL-Event屏蔽寄存器
    reset_intmask( ~(SYNC0_EVENT | SYNC1_EVENT
        | PROCESS_INPUT_EVENT | PROCESS_OUTPUT_EVENT) );
    bEscIntEnabled = FALSE;
    m_pdoinrun = FALSE;
    //禁用SyncManger通道(输入)
    disable_syncmchannel(PROCESS_DATA_IN);
    return 0;
}

//-----
// pdo_stopoutputhandler(): 停止周期性输出通信数据处理
// 返回执行结果
//-----
UINT8 pdo_stopoutputhandler(void)
{
    bEcatOutputUpdateRunning = FALSE;
    return 0;
}

```

参考文献

- [1] IEC61158-1: Industrial communication networks - Fieldbus specifications - Part 1: Overview and guidance for the IEC 61158 and IEC 61784 series.
- [2] IEC61158-2: Industrial communication networks – Fieldbus specifications Part 2: Physical layer specification and service definition – International Standard, Edition 4.
- [3] IEC61158-3-12: Industrial communication networks – Fieldbus specifications – Part 3-12: Data link layer service definition – Type 12 elements.
- [4] IEC61158-4-12: Industrial communication networks – Fieldbus specifications – Part 4-12: -Data-link protocol specification – Type 12 elements.
- [5] IEC61158-5-12: Industrial communication networks – Fieldbus specifications – Part 5-12: - Application layer service definition – Type 12 elements.
- [6] IEC61158-6-12: Industrial communication networks – Fieldbus specifications – Part 6-12: - Application layer protocol specification – Type 12 elements.
- [7] IEC61784-2: Industrial communication networks - Profiles - Part 2: Additional fieldbus profiles for real-time networks based on ISO/IEC 8802-3. Edition 2.
- [8] Ethercat technology group .<http://www.ethercat.org>.
- [9] EtherCAT Technical Introduction and Overview[R]. EtherCAT Technology Group, 2007.
- [10] Beckhoff Automation GmbH. EtherCAT Slave Controller datasheet V1.4. 2008.
- [11] Micrel, Inc. KS8721BL/SL datasheet. 2005.
- [12] IEC 61800-7-1: Adjustable speed electrical power drive systems – Part 7-1: Generic interface and use of profiles for power drive systems – Interface definition.
- [13] IEC 61800-7-200: Adjustable speed electrical power drives systems – Part 7-200: Generic interface and use of profiles for power drive systems – Profile specifications.
- [14] IEC 61800-7-300: Adjustable speed electrical power drives systems – Part 7-300: Generic interface and use of profiles for power drive systems – Mapping of profiles to network technologies.
- [15] 郁极, 尹旭峰. 数字伺服通信协议 SERCOS 驱动程序设计及应用[M]. 北京: 北京航空航天大学出版社, 2005.
- [16] 单春荣, 刘艳强, 郁极. 工业以太网现场总线 EtherCAT 及驱动程序设计[J]. 制造业自动化, 2007 (11) :79~82.
- [17] 刘艳强,王健,单春荣. 基于 EtherCAT 的多轴运动控制器研究[J].制造技术与机床, 2008 (6): 100~103.
- [18] Max Felser. The Fieldbus Standards: History and Structures[EB/OL]. [2006-3-21]. <http://felserr.ch/download/FE-TR-0205.pdf>.
- [19] MAX FELSER. Real-Time Ethernet——Industry Prospective[J]. Proceeding of the IEEE.2005,93(6): 1118-1129.
- [20] Jean-Dominique Decotignie. Ethernet-Based Real_Time and Industrial Communication[J].

- Proceeding of the IEEE.2005,93(6) : 1102-1117.
- [21] 徐皓东, 王宏, 邢志浩. 工业以太网实时通信技术[J]. 信息与控制, 2005,34(1): 60-65.
- [22] 缪学勤.解读IE C61158 第四版现场总线标准[J]. 仪器仪表标准化与计量, 2007(3): 1-4.
- [23] 缪学勤. 试论十种类型现场总线的体系结构[J]. 自动化博览, 2003 (6) :1-6 .
- [24] 梅恪, 沈璞. 关于总线国际标准 IEC61158 的研究报告[J]. 仪器仪表标准化与计量, 2003 (2) :30-34.
- [25] 杨昌琨. IEC61784 实时以太网国际标准简介[J]. 国内外机电一体化技术, 2004, 7 (6) : 57-58.
- [26] 成继勋, 朱红萍. 工业以太网技术的新进展[J]. 自动化仪表, 2004, 25(12): 1-3.
- [27] 缪学勤. 论六种实时以太网的通信协议[J]. 自动化仪表, 2005, 26(4): 1-6.
- [28] WinPcap web site, <http://www.winpcap.org/default.htm>
- [29] 刘文涛. 网络安全开发包详解[M]. 北京: 电子工业出版社, 2005.
- [30] 常发亮, 刘静. 多线程下多媒体定时器在快速数据采集中的应用[J]. 计算机应用, 2003,23(6):177-178.