## 约定

本文档将使用下面的拼写规范：

* *斜体字*用来表示新加入的名词或者是文件名
* 打印机字体用来表示例子代码和命令行输出
* **加粗的打印字体**用来表示命令行输入

数据和地址通常被指定为16进制数值。他们以C语言形式的前缀0x标出（例如：0x88A4）。出发其它提示，地址值都是以字节地址表示的。

函数名称通常带着括号打出来，但是不带有参数。所以，如果一个类似ecrt\_request\_master()这样的函数括号里面没有参数，这不一定说这个函数就不带有参数。

如果需要输入shell命令，会以个dollar符号标示。

$

另外，如果shell程序需要以超级用户的身份被输入，提示符是一个井号：

#

## 1 Igh EtherCAT Master

这一章涵盖了一些EtherCAT主站的基本信息。

### 特点总结

下面的列表给出了主站特点的总结

* 设计为一个Linux 2.6/3.x的内核模块
* 根据IEC 61158-12实现
* 带有本地的EtherCAT兼容的几款常见的Ethernet芯片驱动，还有一个通用的驱动，可以驱动所有的linux 内核支持的芯片。
* 本地驱动在操作硬件时不执行中断
* 对于更多的Ethernet硬件，本地驱动可以简单的通过使用主站模块提供的常用的设备接口来实现（详见4.6）
* 对于任何其他的硬件，可以使用通用驱动。他使用的是linux网络堆中更低的层。
* 主站模块支持多个EtherCAT主站并行运行
* 主站代码通过它的独立式的架构支持任何linux的实时扩展。
* RTAI[11] （包括 通过RTDM的LXRT ），ADEOS，RT-Preempt[12]，Xenomai（包括RTDM），等等。
* 甚至在没有实时扩展时它运行的也很好
* 通常“程序接口”是给想要使用EtherCAT功能的应用程序用的（详见第三章）
* 引入了Domains（域的概念），为了让组织好的进程数据可以通过不同的从站组合任务周期中传送
* 通过几个不同的任务周期处理多个域
* 自动计算进程数据寻址，FMMU和每个域内自己的同步管理配置
* 在几个有限的状态机之下进行通信
* 在拓补结构改变后自动扫描总线
* 在操作时监视总线
* 在操作时自动重新配置从站（例如在掉电之后）
* 支持分布式时钟（查看3.5）
* 通过应用程序接口配置从站DC参数
* 将分布式从站时钟同步到（偏移和零飘补偿）参考时钟上
* 可选择将参考时钟同步到主站时钟上，或者反过来。
* 通过EtherCAT实现CANopen（CoE）
* SDO上传，下载和信息服务
* 通过SDO进行从站配置
* 用户空间和应用程序读取SDO
* 通过EtherCAT实现Ethernet（EoE）
* 通过使用虚拟网络接口透明的使用EoE从站
* 本地支持切换式的或者是轮询式的EoE网络架构
* 通过EtherCAT实现厂商指定（VoE）
* 通过API实现厂商指定的mailbox协议通信。
* 通过EtherCAT实现文件连接（FoE）
* 通过命令行加载和存储文件
* 可以轻松的更新从站的硬件
* 通过EtherCAT实现伺服参数（SoE）
* 根据IEC 61800-7[16]标准实现的
* 存储在启动时写给从站的IDN 配置
* 通过命令行工具读写IDN
* 通过用户空间库在运行时读写IDN
* 用户空间命令行工具“ethercat”（详见7.1）
* 详细的主站，从站，域和总线配置相关的信息
* 设置主站debug等级
* 读/写 别名地址
* 列出从站配置
* 查看过程数据
* SDO上传/下载；列出SDO字典
* 通过FoE加载和存储文件
* SoE IDN 读写
* 从站注册
* 从站SII（EEPROM）的读写
* 控制应用层状态
* 生成从站描述XML和从存在的从站中生成C代码
* 通过LSB兼容实现无缝系统整合
* 通过sysconfig文件实现主站和网络设备的配置
* 主站控制的初始化脚本
* Systemd的服务文件
* 虚拟的只读网络接口，用以监视和调试

### 证书

主站发布的代码是在GNU General Public License（GPL）2.0版本下。非常欢迎其它希望在linux系统下使用EtherCAT的开发者使用主站代码甚至参与到开发中来。

为了可以实现到用户应用程序的静态链接而不是使用主站应用的接口（第三章），用户空间的库函数（7.2小节）是在LGPL,2.1版本下。

## 架构

EtherCAT主站是整合到linux内核内部的。这是很早的设计决定，做这个决定由以下几个原因：

* 内核代码具有很明显更好的实时特性，例如，比用户空间代码更小的延迟。可以预见的，一个总线主站肯定会有很多周期性的工作要做。周期性的工作在内核里通常是由时间终端触发的。当在内核空间时，执行一个处理时间中断函数的时间开销是要更少的，因为不需要将上下文切换到用户空间进程的时间开销。
* 同样可以预见到，主站代码需要直接和Ethernet硬件进行通信。这个无论如何都要在内核空间完成（通过网络设备驱动），这也是另外一个主站代码在内核空间的原因。

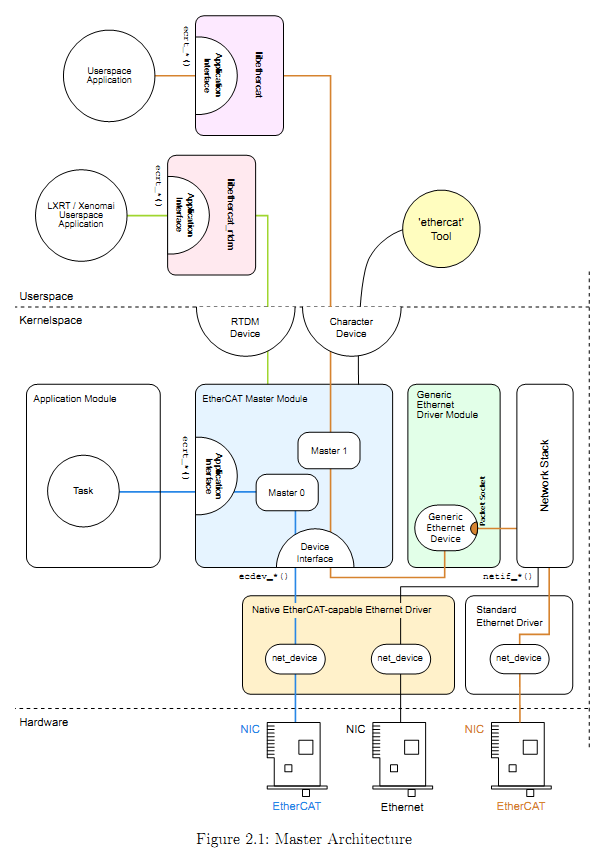
图2.1 给出了一个主站架构的大概外观

主站环境的组成部分描述如下：

**Master Module（主站模块）** 包含了一个或者多个EtherCAT主站实例的内核模块（详见2.1节），“设备接口”（Device Interface）（详见4.6），“应用程序接口”（Application Interface）。

**Device Module（设备模块）** EtherCAT兼容的以太网设备驱动模块，通过设备接口提供他们的设备给EtherCAT主站使用。这些修改过的网络驱动可以并行的处理Ethercat对网络设备的操作和普通的以太网设备。一个主站可以接受一个特定的设备在这之后就可以发送并且接受EtherCAT帧。被主站模块拒绝的以太网设备可以连接到内核的网络堆中。

**Application（应用程序）** 一个使用EtherCAT主站的程序（通常是用来周期性的和EtherCAT从站进行交换数据）。这下程序不是EtherCAT主站代码的一部分，但是还必须要由用户生成或者编写。应用程序可以通过应用程序接口向主站发送请求。如果这步成功了，它就拥有了主站的控制权：它可以提供总线配置和交换过程数据。应用程序可以是内核模块（直接使用内核应用接口）或者也可以是用户空间的程序，用户空间程序就要通过EtherCAT库来使用接口，或者还可以使用RTDM库。



### 2.1 主站模块（Master Module）

EtherCAT主站内核模块ec\_master可以包含多个主站实例。每一个主站等待特定的以太网设备以MAC地址确定。这下地址必须要在加载模块时通过模块参数main\_devices（还有可选项：backup\_devices）指定好。主站实例的个数是根据给定的MAC地址个数来初始化的。

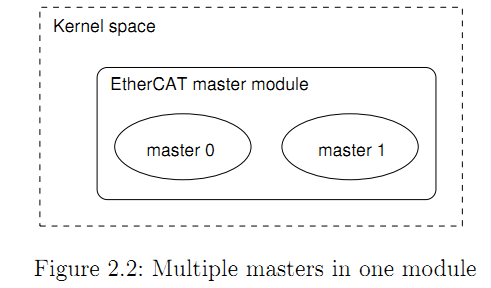
下面的命令加载了一个带有一个主站实例的主站模块，这个实例等待一个MAC地址为的以太网设备。主站可以通过索引0来使用。



多个主站的MAC地址必须要使用冒号来隔开：



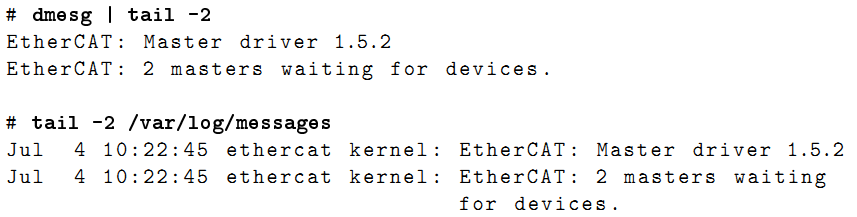
这两个主站可以分别通过他们的索引0和1来定位（看图片2.2）。master索引需要传给ecrt\_master\_request()应用接口函数和ethercat命令行工具的 -- master选项，这个选项默认为0。



**Debug Level （调试等级）**主站模块还有一个参数debug\_level来设定所有主站的初始调试等级（查看7.1.6）

**Init Script （初始化脚本）**在大多数情况下，并不必须要手动加载主站模块和以太网驱动模块。有一个初始化脚本可用，所以主站可以作为一个服务启动（7.4节）。对于由systemd管理的系统来说，也有一个服务文件可用。

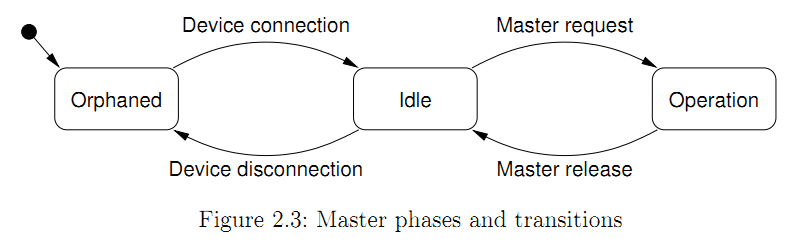
**Syslog （系统日志）** 主站模块会将自己的状态和事件信息输出到内核的环形缓冲器中。这些信息同样会进入到系统日志中。上面的模块加载命令就会产生下面的消息：



主站的输出都会带有EtherCAT前缀，这样在搜索日志的时候会更简单。

### 2.2 主站状态

每一个由主站模块提供的EtherCAT主站，都在几个状态下运行（图2.3）



**Orphaned phase （未认领状态）** 当主站还在等待他的以太网设备连接时，这个模式就会生效。在未连接上设备以前不能使用总线通信。

**Idle phase （空闲状态）** 当主站接受了所有需要的以太网设备，但是还没有任何程序发出请求时，这个状态就会生效。主站运行它的状态机（查看5.3节），这个状态机会自动扫描总线上的从站并且从用户空间接口函数执行挂起操作（例如SDO的操作）。可以使用命令行工具来使用总线，但是由于缺少总线配置，总线上并不会有过程数据。

**Operation phase （操作状态）**主站被一个可以提供总线配置并且交换过程数据的应用程序请求。

### 2.3 Process Data(过程数据)

这一节会介绍几个主站是如何处理过程数据的名词和概念。

**Process Data Image （过程数据映像）**从站通过展示主站所谓的“Process Data Objects”（PDO）提供他们的输入和输出。可用的PDO既可以是通过读取从站E2PROM中的TxPDO和RxPDO SII 列表(以防是固定PDO)还可以是通过读取正确的CoE对象（看6.2小节）来确定的。应用程序可以注册PDO的入口来在周期性的操作时作为数据交换。所有注册的PDO入口的总和定义了“process data image”，它是通过交换使用“逻辑”内存（例如LWR,LRD或者LRW）的datagram（数据报）来引入的[文献2,5.4节]。

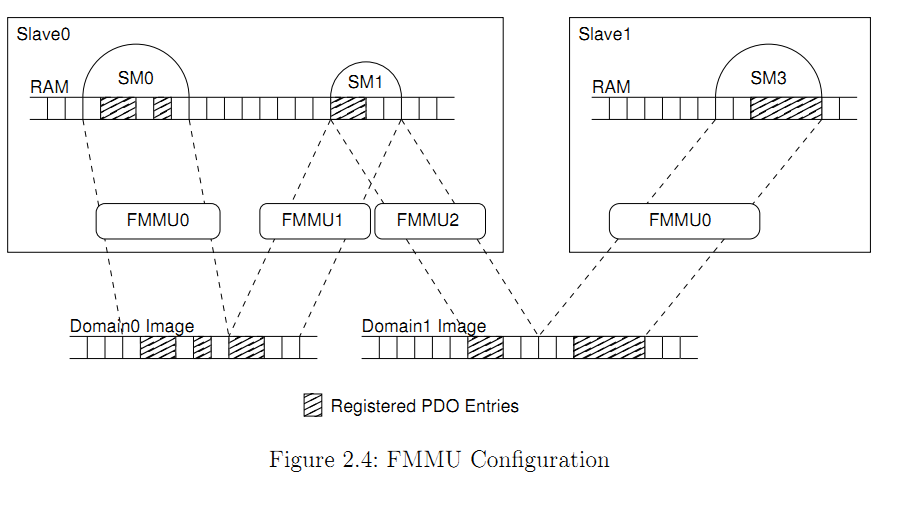
**Process Data Domains （过程数据域）**process data image可以通过建立所谓的“domains”（域）来轻松的实现管理，这个域可以让成组的PDO来进行交换。他们同样会做好管理需要交换的PDO的datagram结构。域对于过程数据交换来说是必要的，所以必须至少有一个。引入域是因为如下原因：

* Datagram的大小是受限制的，因为以太网数据帧是受限制的：最大的数据大小事以太网数据字段大小减去EtherCAT帧头，EtherCAT datagram头和EtherCAT datagram尾：1500 – 2 – 12 -2 = 1484个八位字节。如果process data image的大小超过了这个限制，就需要发送多个数据帧，并且这个image就会由于使用了多个datagram而被分成多个。域的话就可以自动管理这些。
* 并不是每个PDO都必须以一个相同的频率交换：PDO的价值会随着时间而慢慢的改变（例如温度值），所以以一个很高的频率来交换这些值就会变得非常浪费带宽。由于这个原因，可以建立多个域，来组织不同的PDO并且这样就可以分别来交换了。

域的数量是没有上限的，但是每一个域都会在每个涉及到的从站中占据一个FMMU，所以域的最大值是由从站来限制的。

**FMMU Configuration （FMMU 配置）** 一个应用程序可以注册PDO入口来做为数据交换。每一个PDO入口和他的父PDO都是从站物理内存中的一块内存区，这块物理内存是被一个同步管理器[文献2，6.7节]来保护的，这块内存是用来同步用的。为了能让一个同步管理器可以响应一个datagram来使用它的内存，使用到同步管理器所覆盖的最后一个字节是必要的。否则同步管理器不会响应datagram并且不会交换数据。这就是为什么整个同步内存区都必须被包含在process data image中：例如，如果一个特定的从站PDO入口被注册用来和一个特定的域来进行交换，一个FMMU就会被配置来影射整个同步管理器保护的整个内存，PDO就位于这个里面。如果通过一个从站的第二个PDO入口被注册用来在同一块域内交换过程数据，并且它和第一个存在于同一个同步管理器保护内存中，FMMU配置没有被修改，因为目标内存已经是域内process data image中的一部分了。如果第二个PDO如何应该属于另外一个同步管理保护区，这个完整区域也应该被包含进入到process data image域内。

图2.4 给出了一个大致的概念，FMMU是如何被配置来映射物理内存到逻辑process data images的。



## 应用程序接口

应用程序接口给应用程序提供了函数和数据结构来使用EtherCAT主站。完整的接口的文档以Doxygen形式的注释被包含在头文件include/ecrt.h中。它既可以从文件中直接读取，也可以以一种更舒服的HTML文档读取。HTML的生成在9.3节中。

下面的小节涵盖了应用程序接口的一个大致的描述。

每一个应用程序都应该以下面两步来使用主站：

**Configuration** 主站被请求并且提供了主站的配置。例如，域被建立，从站被配置并且PDO入口已经注册（看3.1）

**Operation** 运行周期性代码并且process data 被交换（看3.2节）

**Example Application** 在主站代码中的examples/子文件夹下有一些应用程序的例子。他们是以源代码的形式记录的。

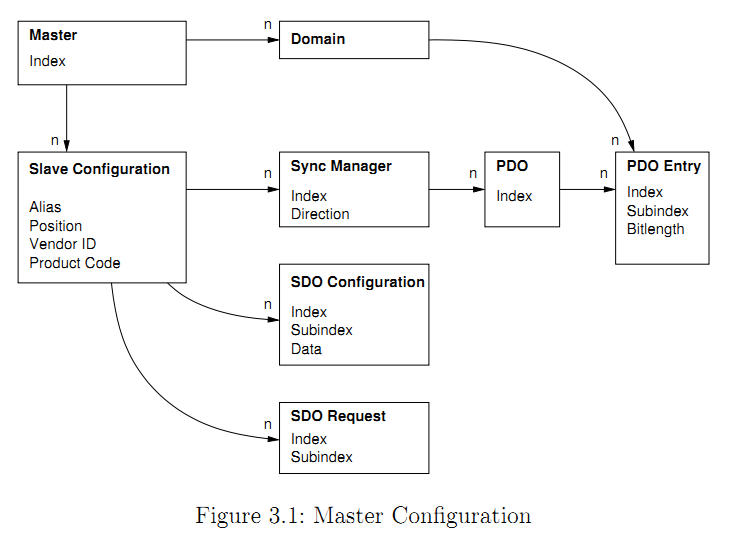
### 3.1 主站配置（Master Configuration）

总线的配置方法是通过应用程序接口来提供的。图3.1 给出了一个对象的示意图，这个对象可以被应用程序配置。

#### 3.1.1 从站配置（Slave Configuration）

应用程序必须要告诉主站关于期望的总线的拓补结构信息。这个可以通过创建“从站配置”来实现。一个从站的配置文件可以被看成是一个期待的从站。当一个从站配置被创建，应用程序提供了总线的位置（看下面），厂商id和产品代码。

当提供了总线配置，主站检查，如果在指定位置有一个带有给定的厂商id和产品码的从站。如果是这种情况，这个从站的配置就是连接到了总线上的真正从站上了（“attached”）并且这个从站会根据应用程序提供的设定配置好。从站配置的状态可以通过应用程序接口查询也可以通过命令行工具查询（7.1.3小节）。



**从站位置 （Slave Position）**从站位置必须被指定为“alias”（别名）和“position”（位置）的一个元组。这就可以 通过绝对总线位置，或者一个保存的叫做“alias”（别名）的标示符，或者两者的混合的方式来定位从站。这个alias就是在从站的E2PROM中保存的一个16位的值。它可以通过命令行工具来修改（查看7.1.2小节）。表格3.1 表明了这些值是如何被解释的。

|  |  |  |
| --- | --- | --- |
| 别名 | 位置 | 解释 |
| 0 | 0-65535 | 位置寻址。位置参数被解释为总线上的绝对环形位置。 |
| 1-65535 | 0-65535 | 别名寻址。位置参数被解释为一个相对的位置，它跟在 |

图3.2 展示了一个从站配置是人利被连接的例子。有一些配置被连接了，而其他的没有被连接。下面的列表给出了从头一个从站配置开始的原因。

1. Alias为0就意味着使用简单的位置寻址。从站1存在并且厂商id和产品代码和预期值相匹配。
2. 虽然带有位置0的从站被找到了，但是产品代码不匹配，所以配置没有给连接上。
3. alias为非0的值，所以使用alias寻址。从站2是第一个带有0x2000别名的从站。因为位置值为0，所以使用相同的从站。
4. 没有带有给定别名的从站，所以配置没有被连接。
5. 从站2又是第一个带有0x2000别名的从站，但是位置现在为1了，所以从站3被连接。

如果主站源代码被配置为 –enable – wildcards，那么0xffffffff就会匹配任何的厂商ID或者产品代码。

### 3.2 周期性操作

为了要进入周期操作模式，主站必须被“激活”（activated）来计算process data image 并且首次应用总线配置。在激活之后，应用程序就负责发送和接收数据帧了。在激活以后，配置就不能被修改了。

### 3.3 VoE句柄（VoE Handler）

在配置阶段，应用程序可以创建6.3小节中描述的VoE mailbox协议的句柄。一个VoE句柄总是属于一个特定的从站配置，所以创建函数就是从站配置的一种方法。

VoE句柄管理着VoE数据和用来传输并接受VoE消息的datagram。它包含了传递VoE消息所必要的状态机。

VoE状态在同一时刻只能处理一个操作。所以，同一时刻只能要么执行一个读或者要么执行写操作。在操作被启动后，句柄必须在它被完成之前一直周期执行。在完成之后，可以检索到操作的结果。

VoE句柄具有一个自己的datagram结构，它是用来标记每次执行过后的交换过程的。所以应用程序就可以决定，在发送对应的EtherCAT帧之前决定需要执行多少次句柄。

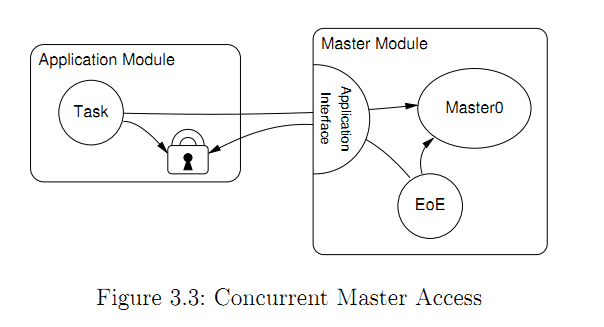
要获取更多的关于VoE句柄的信息可以查看应用程序接口函数和examples文件夹下提供的例子程序。

### 3.4 使用并发的主站

有些情况，一个主站有几个实例，例如当一个应用程序在做周期性的数据交换，并且还有EoE从站请求从内核交换以太网数据。由于这个原因，主站是一个共享的资源，并且使用它必须是由顺序的。这通常是通过锁住信号量来实现的，或者是使用其他方法来保护关键资源。

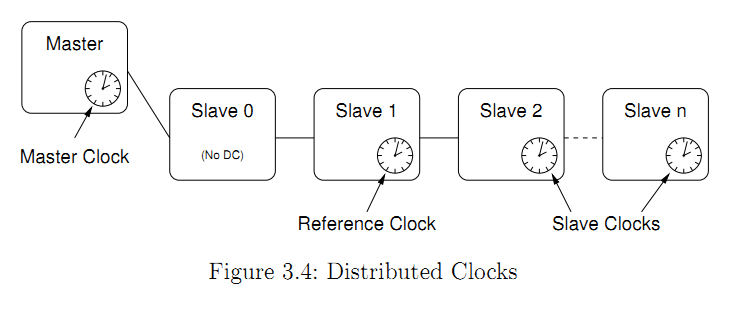
主站自己并没有办法提供锁机制，因为它并没有办法知道正确锁的类型。例如如果应用程序是在内核空间并且使用RTAI功能函数，通常内核 并不足以应对。所以，这里做了一个重要的设计决定：预约了主站的应用程序必须拥有完整的控制，所以他必须它必须负责提供正确的锁机制。如果另外一个实例想要使用这个主站，它必须通过回调函数来请求总线的使用，这必须由应用程序来提供。此外如果应用程序认为现在这个时刻比较糟糕它还可以拒绝链接到主站上。

图3.3 显示了一个例子，说明了两个进程是如果共享一个主站的：应用程序的周期性任务使用主站交换数据，而主站内部的EoE进程使用它来和EoE从站进行通信。他们都不时的使用总线，但是EoE应用通过“请求”应用来完成这个过程。这样，银城就可以使用正确的锁机制来避免同一时刻来使用总线。查看应用程序接口文件如何使用这些回调函数。



### 3.5 分布式时钟（Distributed Clocks）

从1.5版本以后，主站就EtherCAT“分布式时钟”特性。将从站的总线上的时钟同步到“参考时钟”（reference clock）（就是第一个支持DC的从站的本地时钟）是可能的并且还可以将参考时钟同步到“主站时钟”（master lock）上。总线上所有其他的时钟（在参考时钟之后的）都被认为是“从站时钟”（slave clocks）

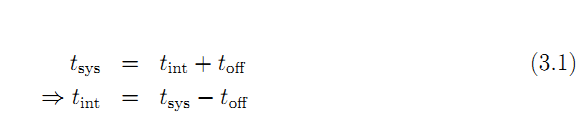


**本地时钟（Local Clocks）** 任何支持DC的EtherCAT从站都有一个注册的以那秒为单位的本地时钟。如果从站上电了，时钟从0开始，这就意味着从站在不同的时刻上电，他们的时钟就会有不同的值。这些“偏移”必须要用分布式时钟的机制来补偿掉。另外，这些时钟也不是以一个同样的速度在运行，因为使用的石英单元具有很自然的频率差。这个差距通常非常小，但是在很长的时间以后，这个误差就会累计并且本地时钟之间的差距就会增长。这个时钟“漂移”（drift）也要使用DC机制来补偿掉。

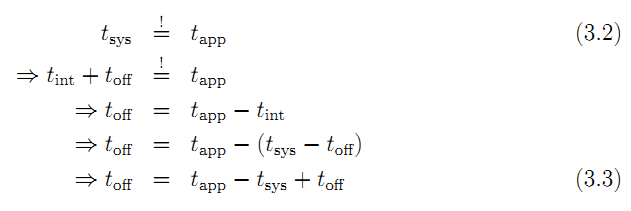
**应用时间（Application Time）**通常总线的时间基准是必须由应用程序提供的。这个应该程序时间被用来：

1. 配置从站的时钟偏移（看下面）
2. 为了要编辑从站的开始时间用来产生同步脉冲（看下面）
3. 同步参考时钟到主站时钟（可选）

**偏移补偿（Offset Compensation）** 为了要补偿偏移量，每一个从站都提供一个“系统时间偏移”寄存器，它被用来加到内部时钟值 上来获得“系统时间”（System Time）：



主站读取两个寄存器的值来计算新的系统时间偏移，那么结果就是系统时间应该和主站的程序时间相匹配：



由于读和写寄存器产生的微小的时间长会在后面被漂移补偿补偿掉。

**漂移补偿（Drift Compensation）**由于在每个支持DC的从站里的特殊机制，漂移补偿是可以实现的：一个对“系统时间”寄存器会造成内部时间控制循环和写入时间进行对比（会减去编程传送延迟，看下面）来获得当前的系统时间。计算出来的时间差会被用来作为一个时间控制器的输入，这个控制器就会将本地时钟速度调快一点或者慢一点，根据时间差的符号。

**传输延时（Transmission Delays）**以太网帧需要一个很小的时间来从一个从站到另外一个从站。这就会造成传送延迟时间在总线上累加并且可以达到毫秒级，并且也是必须在做漂移补偿的时候考虑。支持DC的EtherCAT从站提供了一个机制了测量传送延迟：每4个从站端口就有一个接受时间寄存器。一个对接收时间寄存器端口0的写操作就会触发开始计时并且一旦一个帧被对应的端口接受了就会锁住当前的系统时间并存储在一个接受时间寄存器里。主站可以读出相对的接受时间，然后计算在从站之间的时间延迟（使用它的总线拓补结构知识），并且最终计算出每个从站对于参考时钟的时间延迟。这些值都会编程到从站的传输延时寄存器中。用这种方式，漂移补偿可以达到纳秒级的同步效果。

2：本地从站时钟会每10纳秒做9,10或者11纳秒的增量。

**检查同步性** 支持DC的从站提供32为的“系统时间偏差”寄存器，地址为0x092c，上一次漂移补偿的系统时间差就以纳秒为单位并且带有符合等级编码方式（3）存放在这个地址。为了要检查总线同步性，系统时间偏差寄存器也可以通过命令行周期性的读取。



同步信号 同步的时钟只是总线上同步事件的前提。每个支持DC的从站都提供两个“同步信号”，这两个信号可以通过编程来产生事件，这会例如造成从站应用程序在一个特定时间锁定它的输入。同步事件可以只产生一次也可以周期性产生，这依赖于对于从站应用程序来说哪种才是有意义的。编程同步信号其实就是设定所谓的“分配激活”（AssignActivate）字和同步信号周期还有漂移时间。分配激活字是从站指定并且必须要由XML从站描述中读取（Device->Dc），在这个文件里还可以找到典型的同步信号配置“操作模式”（OpModes）。

## 以太网设备（Ethernet Devices）

EtherCAT协议是基于以太网标准的，所以主站是依赖于标准的以太网硬件来与总线进行通信的。

Device这个名词是以太网网络接口硬件（Ethernet network interface）的同义词。

**本地以太网设备驱动（Native Ethernet Device Drivers）**有本地以太网设备模块来处理以太网硬件，主站可以使用它来链接到EtherCAT总线上。他们通过设备接口让主站模块可以使用以太网硬件并且必须具备将以太网设备准备给EtherCAT（实时）操作还可以做“普通”的使用内核网络堆的操作。这样做的好处是主站几乎可以直接操作硬件，这就会提供一个非常高的表现。缺点是，原版以太网驱动必须要有一个兼容EtherCAT的版本。

**通用以太网设备驱动（Generic Ethernet Device Driver）**从1.5版本以后的主站，包里面就带有一个通用的以太网设备驱动模块（见4.3），它使用底层网络堆来和硬件连接。有点事，任意的以太网设备都可以用来做EtherCAT操作，并不依赖于实际的硬件驱动（所以所有的以太网驱动都可以不用修改被支持）。缺点就是，这种方法并不支持类似RTAI这样的实时扩展，因为linux网络堆被寻址。并且这样的行为要比本地方法差一点，因为以太网数据帧通过网络堆。

### 4.1 网络设备基础

EtherCAT依赖于以太网硬件并且主站需要一个物理以太网设备来和总线通信。所以，对于linux如何处理网络设备和他们的驱动是非常必要的。

**网络驱动任务（Tasks of a Network Driver）**网络设备驱动通常处理低于OSI模型两层的内容，就是物理层和数据链路成。网络设备自己在本地处理物理层的问题：它表示硬件要连接到中间层并且要发送并且发送数据，物理层协议描述。

网络设备驱动负责从内核网络堆中获取数据并且将数据运送到硬件，硬件会实现物理传输。如何数据分别被硬件接受，驱动被通知（通常意味着会有一个中断）并且必须要从硬件内存中读取数据并且将它发送到网络堆中。还有几个任务，网络驱动需要处理的，包括队列控制，统计还有设备依赖特点。

**驱动启动（Driver Startup）**通常，一个驱动在模块加载的时候搜索兼容的设备。对于PCI驱动来说，这步是通过扫描PCI总线并且检查已知的设备ID。如果设备被找到，分配数据结构然后设备就进入操作。

**终端操作（Interrupt Operation）** 网络设备通常提供一个硬件中断，这个中断用来通知驱动接收到了数据帧并且传送成功，没有错误。驱动需要注册一个中断服务周期处理（ISR），这个处理会在每次硬件发送这样一个事件的信号时都执行一次。如果设备自己抛出了中断（多个设备可以共享一个硬件中断），这就是为什么中断必须要通过读取设备中断注册表来确定的原因。例如，如果接受到数据帧的标着被设定了，数据就必须被从硬件中拷贝到内核内存并且传递给网络堆中。

**Net\_device结构** 驱动为每一个和网络堆通信的设备注册一个net\_device结构并且创建一个“网络接口”。例如一个以太网设备，这个接口就会使ethX,这个X是一个内核在注册时分配的数字。Net\_device结构通过回调函数接收事件（要么从用户空间，要么从网络堆），这个回调函数必须要在注册之前就设定好。并不是每个回调都是必须的，但是对于比较合理的操作是在任何情况下面都应该需要的：

Open() 这个函数是当网络通信被开始时调用的，例如在用户控件执行一个命令ip link set ethX up时。数据帧接受必须被驱动使能。

Stop() 这个回调函数的目的是“关闭”设备，例如，让硬件停止接受数据帧。

Hard\_start\_xmit() 这个函数在每次数据帧需要被发送时被调用。网络堆将数据帧作为一个指向sk\_buff结构的指针来传递（“socket buff”，），它必须在发送后被free。

Get\_stats（） 这个调用必须要返回一个指向设备net\_device\_stats结构的指针，它永远要用数据帧填满。这就意味着，每一次有帧被接收，发送或者发生错误，在这个结构中的对应的结构就会增加。

真正的注册是由register\_netdev()调用完成的，移除注册时由unregister\_netdev（）函数完成的。

**Netif接口** 所有的直接接口中的其他链接->网络堆都是通过netif\_\*()调用来实现的。例如，设备成功打开以后，必须要通知网络堆，这样它就可以传递数据帧给接口了。这是通过调用netif\_start\_queue()来实现的。在调用这个函数以后，网络堆可以调用hard\_start\_xmit()回调函数。还有网络驱动通常管理着一个数据站传递队列。如果这个队列被塞满了，必须要通知网络堆停止再传送数据帧了。这要通过调用netif\_stop\_queue()。如果已经发送了一些数据帧了，并且还有足够的空间存储新的数据帧，这个可以使用netif\_wake\_queue()函数来 通知。另外一个重要的调用时netif\_receive\_skb()：它传递一个数据帧到网络堆，这个帧刚刚被设备接收。帧数据必须要被包含在一个叫做“socket buff”的缓冲区里。

**Socket Buffer** socket buffer是整个网络堆中最基本的数据类型。他们作为网络数据的容器存在并且还可以快速的添加数据头和尾，或者是再将头和尾再次去掉。所以socket buffer 包含一个分配的缓冲区还有几个指向缓冲区开始（head），数据开始位置（data），结尾位置（tail）还有缓冲区结束位置（end）的指针。另外，socket buffer保持了网络头信息还有（如何受到了数据信息）一个指针指向net\_device。有几个函数用来创建socket buffer（dev\_alloc\_skb（）），可以从头（skb\_push()）或者尾（skb\_put()）添加数据，也可以从头（skb\_pull()）或者从尾部（skb\_trim()）删除数据，或者删除整个buffer（kfree\_skb()）。Socket buffer从一层传递到另外一层，并且在最后一次使用它的那一层给它释放。以发送为例，释放必须要被网络驱动来完成。

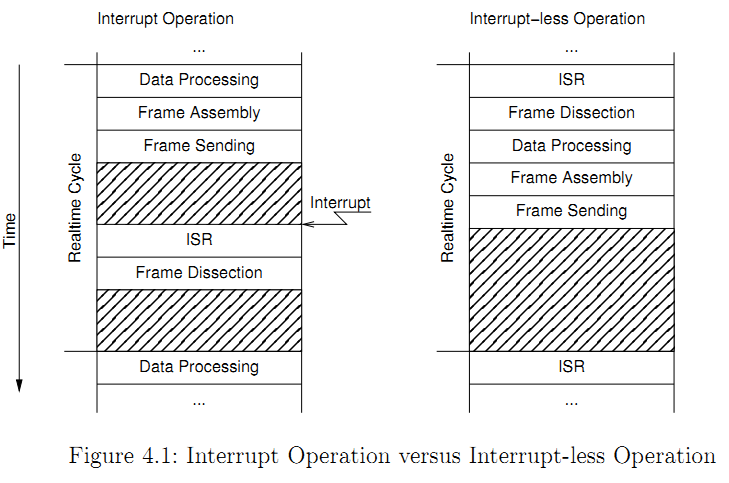
### 4.2 本地EtherCAT设备驱动

当使用本地以太网驱动来做EtherCAT功能时，对以太网硬件有几个要求。

**专用的硬件** 为了表现与实时的目的，EtherCAT主站需要直接并且唯一的使用以太网硬件。这就预示着以太网硬件必须不能像往常那样链接到内核堆中，因为内核会尝试将 它作为一个普通的以太网设备来使用。

**少中断操作** EtherCAT数据帧穿越过整个EtherCAT逻辑环形缓冲区并且最后发送回给主站。整个通信过程是高度确定的：一个数据帧被发送并且再次接受都是经过一个固定的时间的，所以就没有必要通知驱动接受到数据帧了：主站可以询问硬件接受到的数据帧，如果它认为应该已经收到数据帧了。

图4.1 显示了两个工作流分别在有和没有中断的情况下周期性的传递并且接受数据帧。



在左边的工作流“中断操作”，上一个周期的数据被第一个处理并且和新的datagram组成一个新的帧，然后它被发送。周期性工作到这就完成了。稍后，当硬件再次接收到数据帧时，会触发一个中断并且执行ISR。ISR就会从硬件获取数据帧并且开始数据帧分解：datagram会被处理，所以数据已经为下个周期准备好了。

在右边的工作流“少中断操作”，没有使能硬件中断。替代的，硬件会被主站通过执行ISR控制。如果同时数据帧已经被接收了，他就会被分解。到这，情况和左边的工作流开始时都是一样的：接收到的数据被处理并且组成一个新的帧并且发送。这个周期后面就没有工作需要做了。

更倾向于使用无中断操作，因为硬件中断对于改进驱动的实时性表现是没有益处的：他们的随机发生性会造成增加抖动。还有，如果使用了一个实时扩展（RTAI），还需要做其它的工作来提高中断的优先级。

**以太网和EtherCAT设备** 另外一个就是linux处理同样类型设备的方法。例如，一个PCI驱动扫描PCI总线上它能处理的设备。然后它就会将自己注册为对所有找到的设备负责。问题是，没有修改过的驱动是不能因为接下来这个设备会被做EtherCAT通信而忽略这个设备的。必须有一种方法来处理多个相同类型设备，这些设备其中一个是为EtherCAT所预留的，而另外一个是被当做一个普通的以太网设备。

由于所有的这些原因，作者决定唯一可以接受的解决方法是用一种方法修改标准的以太网设备驱动，让他们既可以保持他们正常的功能，但是具备将一个或者多个设备当场EtherCAT设备处理的能力。

下面是这种方法的有点：

* 不需要告诉标准驱动忽略特定的设备
* 可以用一个驱动来给EtherCAT和非EtherCAT设备用
* 没有必要从头实现一个网络驱动，并遇到前面开发者已经解决的问题

选择的方法有下面的缺点：

* 修改的驱动变得更复杂了，因为它必须处理EtherCAT和非EtherCAT设备。
* 在驱动代码里增加了很多不同的地方。
* 标准驱动的修改和bug处理必须要是不是的加入到EtherCAT版本的驱动中。

### 4.3 通用的EtherCAT设备驱动

因为有方法可以使整个linux内核都做实时操作，不适用本地的EtherCAT以太网设备驱动而是使用linux网络堆也是可行的。图2.1显示了“Generic Ethernet Driver Module”，它连通过网络堆连接到本地以太网设备。内核模块名字为ec\_generic并且可以在主站模块后面作为一个本地EtherCAT的以太网驱动加载。

通用设备驱动扫描网络堆中的意境被以太网设备驱动注册的接口。它提供给EtherCAT主站所有可用设备。如果主站接受了一个设备，通用驱动创建一个包 socket（查看linux的 man 7 packet），其中socket\_type设定为SOCK\_RAW，绑定到那个设备上。所有这个设备接口的函数（4.6节）就会在socket上操作。

下面是这种解决方案的优点：

* 任何以太网硬件，只要是linux以太网驱动包含的，就可以被用作EtherCAT。
* 不需要对真正的以太网驱动做修改

通用方法有下面的缺点：

* 这种方法的表现会比使用本地驱动方法差一点，因为数据帧必须要穿过网络堆更低层。
* 不能在in-kernel实时扩展，例如RTAI上使用通用驱动，因为网络堆代码使用冬天内存分配和其它一些东西，这会造成系统在实时环境中卡死。

**设备激活** 为了通过socket发送并接受帧，EtherCAT链接的那个socket必须要被激活，否则所有的帧都会被拒绝。激活必须在主站模块加载之前执行并且可以以集中方式执行：

* Ad-hoc，使用命令 ip link set dev ethX up(或者更老的ifconfig ethX up)，
* 配置，根据不同的发布版本，例如在某些openSUSE和其它系统使用ifcfg 文件（/etc/sysconfig/network/ifcfg-ethX）。如果EtherCAT主站在系统启动时候开启回是一个比较好的主意。由于以太网设备应该只是被激活，但是不应该分配IP地址等等其它，使用STARTMODE = auto配置就足够了。

### 4.4 提供以太网设备

在加载了主站模块后，还有几个模块需要被加载来给主站提供设备。主站模块知道要从模块参数中选择哪些设备。如何是使用初始化脚本来启动主站，需要使用的驱动和设备可以声明在sysconfig文件中（看7.4.2）

提供以太网设备的模块可以是

* 本地EtherCAT兼容的网络驱动模块或者是
* Generic EtherCAT设备驱动模块

### 4.5 冗余

冗余总线操作意味着，从主站到从站具有多于一个的以太网链接。每一个主站链接都会发送过程数据交换datagram，所以交换还是完整的，甚至总线在中间某处断掉了。

完整的冗余总线操作的前提是，每一个从站都至少有一个主站可以连接上。这样有一个链接失败（i.e总线断线）就不会导致数据不完整。两个以太网设备的话，两个都报错就处理不了了。

冗余是在配置的时候使用 - -with – devices配置选项并且ec\_master内核模块使用backup\_devices参数（2.1），或者在sysconfig文件中使用正确的变量MASTERx\_BACKUP(7.4.2)

在任何以太网连接的拓补结构改变后，都会进行总线扫描。应用程序接口和命令行工具（7.1）都有方法可以询问冗余操作的状态。

### 4.6 EtherCAT设备接口

对于关于主站模块的那个小节的一个期望是让读者可以理解网络设备驱动模块链接一个设备到特定的EtherCAT主站的一种方式。

主站模块为网络设备驱动提供一个“设备接口”。为了使用这个接口，网络设备驱动模块必须包含devices/ecdev.h这个头文件，还有EtherCAT主站代码。这个头文件给EtherCAT设备提供了一个函数结构。所有的设备接口函数的名字都带有一个ecdev前缀。

### 4.7 本地驱动打补丁

这一节会描述，如何让一个标准的以太网驱动可以为EtherCAT使用，使用本地方式。不幸的是，并没有标准的程序来让以太网驱动可以为EtherCAT主站使用，但是有几个通用的技术。

1. 首先一个简单的原则是，netif\_\*()调用必须要避免被所有的EtherCAT设备调用。由之前提到过的那样，EtherCAT设备并不链接到网络堆中，并且因此必须禁止调用它的接口函数。
2. 另外一个重要的事情是，EtherCAT设备应该以无中断的方式操作。所以必须要避免任何在硬件层面注册中断句柄和使能中断的函数调用。
3. 主站并不是每一次发送操作都使用一个新的socket buffer：相反的在主站初始化时会分配一个固定的。这个socket buffer在每次发送操作时都会被塞进去一个EtherCAT帧并发送给hard\_start\_xmit()回调函数。由于这个socket buffer是必要的，所以它不会像 通常那样被驱动所释放（free）。

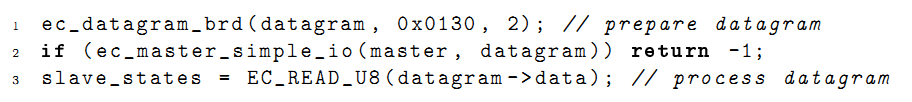
一个以太网驱动通常处理几个以太网设备，每个都是由一个带有一个连接到依赖于驱动的数据上的priv\_data字段的net\_device结构来描述的。为了要将普通的以太网设备和被EtherCAT主站使用的设备区分开，驱动使用的私有数据可以被展开成为一个指针，如果设备是被主站使用的，它指向一个ecdev\_offer()返回的ec\_device\_t对象，否则这个指针是0。

RealTke RTL-8139 Fast Ethernet driver是一个“简单的”以太网驱动并且可以作为一个给新驱动打补丁的例子。在文件devices/8139too-2.6.24-ethercat.c中搜索“ecdev”可以找到一些有趣的内容。

## 状态机

许多EtherCAT主站的部分都是作为一个有限状态机实现的（FSMs）。虽然这将导致某些方面高度复杂，但是却打开了很多新的可能性。

下面的简短例子代码表示了如何读取所有的伺服状态并且还说明了什么是严格的顺序编码：



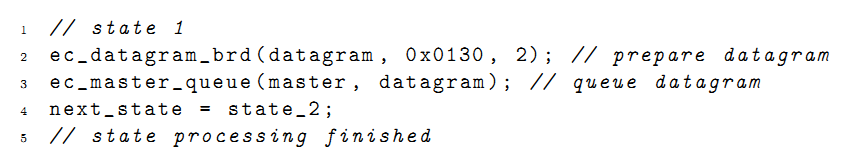
ec\_master\_simple\_io（）函数提供了一个简单的接口来同步发送一个单一的datagram并接受结果。内部来讲，它将制定的datagram排队，调用ec\_master\_send\_datagrams（）函数来发送带有队列的datagram的数据帧然后等待接受它被接收。

这个顺序的方法非常简单，反映出来就只有3行代码。不过缺点是，主站在 它等待datagram被接受时都会被阻塞住。在只有一个实例使用主站时，这没啥难度，但是如果多个实例想要（并行的）使用主站，那就不可避免的需要想一种另外的队列模型。

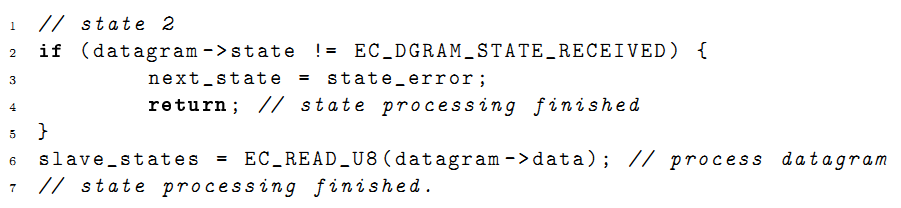
使用主站的使用也必须是有顺序的，因为有多于一个的实例想要同步的发送和接收datagram。使用现有的方法，这就会造成一种等待每个实例的状态，这种状态是不能被接受的尤其是在实时环境下，因为浪费掉的巨量的时间。

一种可能的解决方法时，让所有的实例顺序的将他们的datagram都加入队列，然将控制权给到下一个实例而不是等待datagram被接收。最终，总线IO被一个更高的实例完成，这就意味着所有进队列的datagram都被发送并且接受了。下一步就是再次执行所有的实例，这时他们就会处理他们接收到的datagram并且发布新数据。

这种方法就导致所有的实例都必须在将控制权给会到更高的实例时，要保持他们的状态。在这种情况下很明显就必须要使用一种有限状态机模型。5.1节就会介绍一些使用理论的内容，而下面列出的内容展示了将上述例子编程为状态机的代码的基本方法：



在所有实例执行完他们的当前状态并且将他们的datagram放入队列后，这些就会被发送并且接收。然后对应的下面的状态就会被执行：



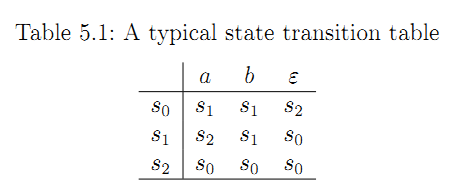
查看5.2节中主站代码中使用的状态机编程的理念。

### 状态机理论

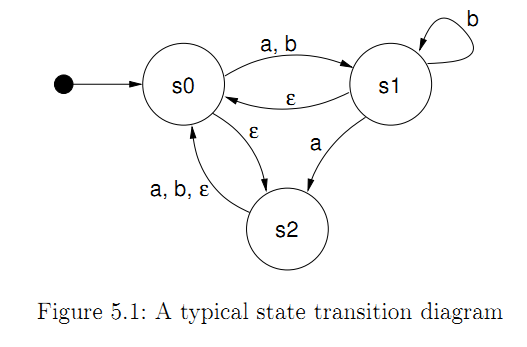
一个有限状态机是一个有着带有输入和输出行为的模型，这里输出不单单依赖于输入，而是历史输入。数学上有限状态机的定义是一个六元组（），带有

* 输入字母∑，∑≠ ， 包含所有的输入标志。
* 输出列表 ，包含所有的输出标志。
* 状态集合，
* 初始状态集合 
* 输出函数为ω

状态机函数 通常由一个状态转换表声明，或者是被一个专题转换图声明。转变表提供了一个状态机行为的矩阵。矩阵的行对应状态（S={s0,s1,s2}）并且列对应输入信号。表中某一特定的i行j列的内容就表示下一个状态（还有可能的输出），某一个特定的输入信号被读入到状态si中。



同一个例子的状态图标就是像图5.1这样。状态是由圆形或者椭圆形表示，而转换画成在他们之间的箭头。靠近转换箭头的就是要完成转换必须要满足的条件。初始状态是由一个实心黑圆圈带着一个箭头指向对应的状态表示出来的。



**确定于非确定状态机** 状态机可以是确定的，就是说对于一个状态和输入，有且只有一个下面的状态。这种情况下，状态机只有一个确定的开始状态。非确定状态机对于一个状态-输入可以有多个转换组合。在后面这种情况下就有一组初始状态。

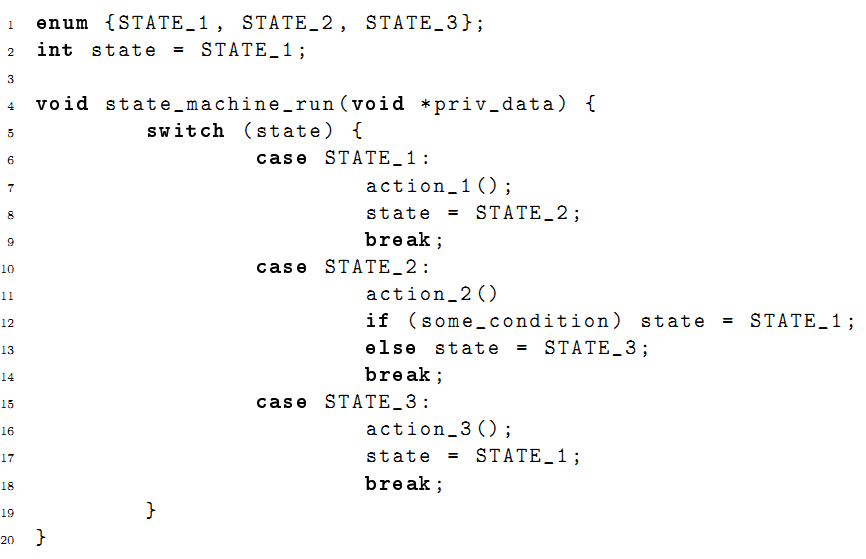
**Moore 和Mealy 模型** 在所谓的Moore模型和Mealy模型之间是有区别的。从数学上讲，区别在输出函数ω上：如果它只依赖于当前状态（），状态机对于“Moore 模型”。否则，如何ω是一个状态和输入表的函数（），这个状态机对应的就是“Mealy 模型”。在大多数情况下，Mealy状态机都是更加实用的解决方案，因为这样的设计让状态机可以有最少的状态。实际中，通常实用两种模型的混合。

**状态机的误解** 有一种现象叫做“状态爆炸”，通常是在复杂环境中正常实用状态时接受了一个反向的参数。这里必须要提一下，这是一个误解。状态爆炸通常是因为错误的状态机设计造成的：通常的错误是将一个状态的所有输入当前值都保存，或者没有将一个复杂的状态机分解成简单的子状态机。EtherCAT主站使用几个状态机，他们是分层次执行并且作为一个子状态机互相服务。下面会介绍这个。

### 主站状态机

这一节会引入主站中实现状态机使用的技术。

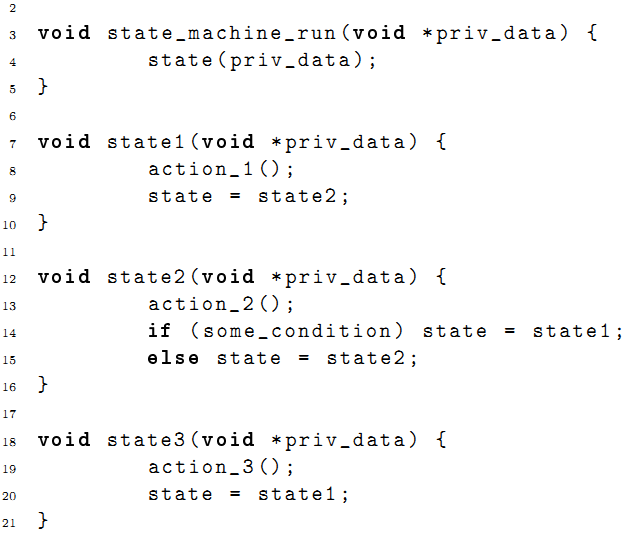
**状态机编程** 这是一种特定的方式以C代码实现状态机。一种明显的方式是通过一个大的case语句来实现不同的状态和行为：



对于小的状态机来说，这是一个选项。缺点是，随着状态的增加，代码很快就会变得复杂并且新增加的case分支每次运行都会被执行。并且，浪费了很多缩进空间。

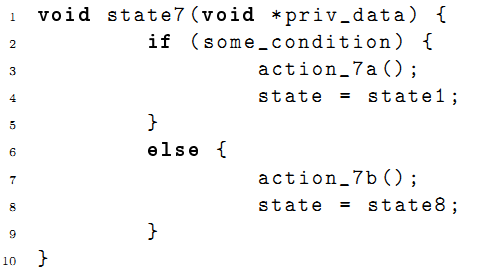
主站中使用的方法时在各自的函数中实现每一个状态并且使用一个函数指针保存当前状态函数。





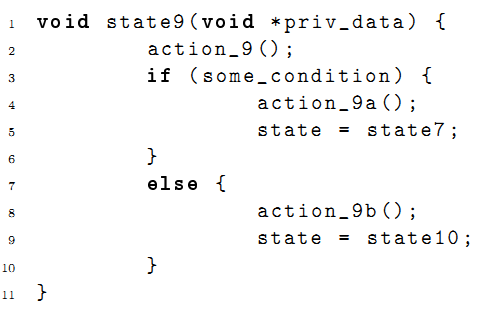
在主站代码中，所有状态机的状态指针都被搜集在一个ec\_fsm\_master\_t类的对象当中。这是个优点，因为每一个状态总有一个实例可用并且可以依着需要从头开始。

**Mealy 和 Moore** 如果仔细看上面的列表，可以见执行的行为只依赖于当前的状态（状态机的“输出”）。这和5.1节介绍的“Moore”模型是一样的。如前面提到的那样，“Mealy”模型提供了更高的灵活性，它可以在下面的列表中看到：



状态函数依赖于状态转换来执行动作。

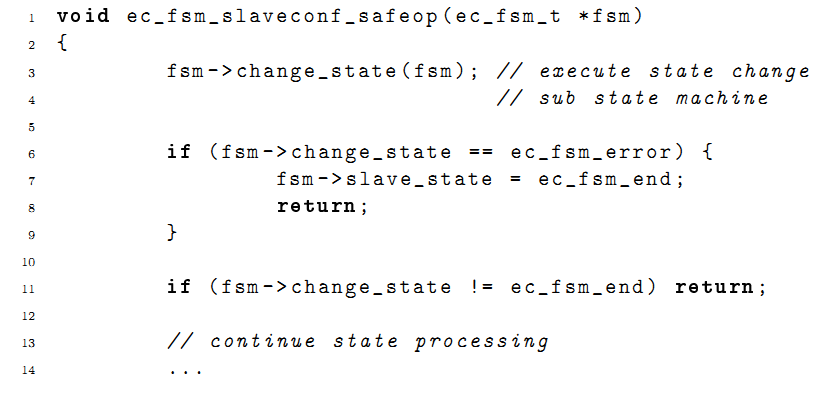
最灵活的修改是依赖于状态执行一些行为，接下来的行为依赖于转换：



这种模型通常用在主站代码中。它将两种方法的有点结合起来。

**使用子状态机** 为了避免带有过多的状态，一些EtherCAT主站状态机已经被编码成为子状态机。这会帮助封装相关的工作流并且避免5.1节中谈过的“状态爆炸”现象。如果主站的使用一个大的状态机，状态的数量会使实际数量的几倍多。这会将负责等级增加到无法管理的等级。

**执行子状态机** 如果一个状态开始执行一个子状态机，它通常还保持在一个状态一直到子状态机结束。这通常完成如下列表，它是从从站配置状态机代码中拿出来的：



change\_state 是状态改变状态机的状态指针。状态函数，指针指向的，被执行….

… 直到状态机带着错误执行完毕…

…. 或者直到状态机以结束状态执行完毕。直到这时，“更高一级”的状态机一直保存当前状态并且在下一个循环技术执行子状态机。

**状态机描述** 下面的小节描述在EtherCAT主站中使用的每个状态机。状态机文字性的描述包含了对应状态机转换表中转换的参数，用的是一个箭头后面带着接下来的状态标记的。由于不重要的错误（例如，不是从中站响应的错误）产生的转换，没有被显示的描述。这些转换在图中以虚线画出。

### 主站状态机

主站状态机是在主站线程上下文中执行的。图5.2显示了它的转换表。它应该是：

**总线监视** 监视总线拓补结构。如果它改变，总线被重新扫描。

**从站配置** 监视应用层从站的状态。如果从站不在他应该处在的状态，从站被重新配置。

**请求处理** 处理请求（从应用程序发出的或者是从外部源发出的都处理）。一个请求就是主站应该异步处理的工作，例如使用SII，使用SDO，或者类似的。

### 从站扫描状态机

从站扫描状态机，可以在图5.3中看到，它引领整个读取目标从站信息的整个过程。

**节点地址** 从站节点地址被设定，这样后面的操作都可以通过节点来定位。

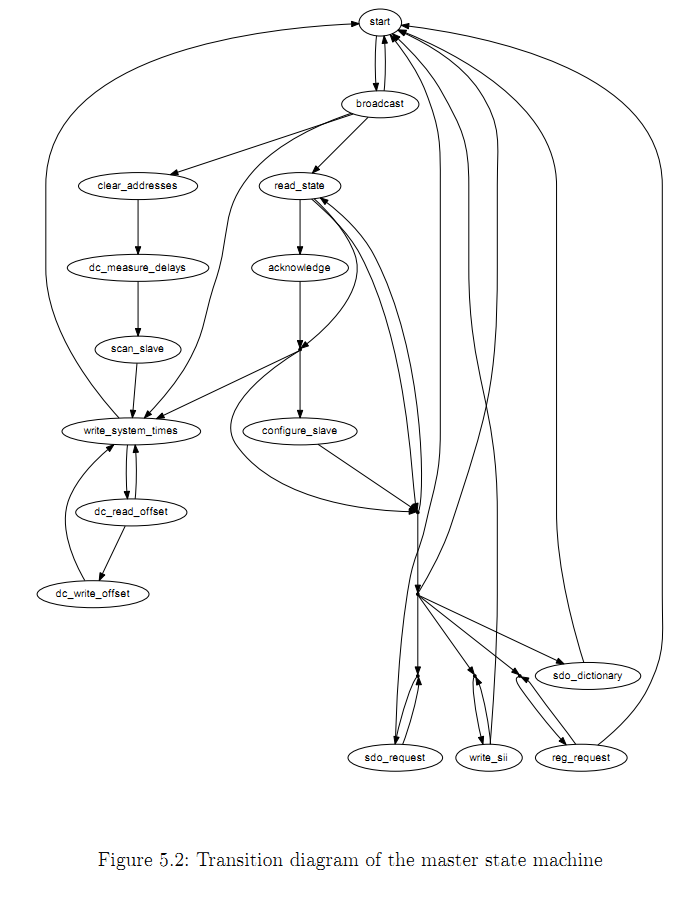
**AL 状态** 初始应用程状态被读取。

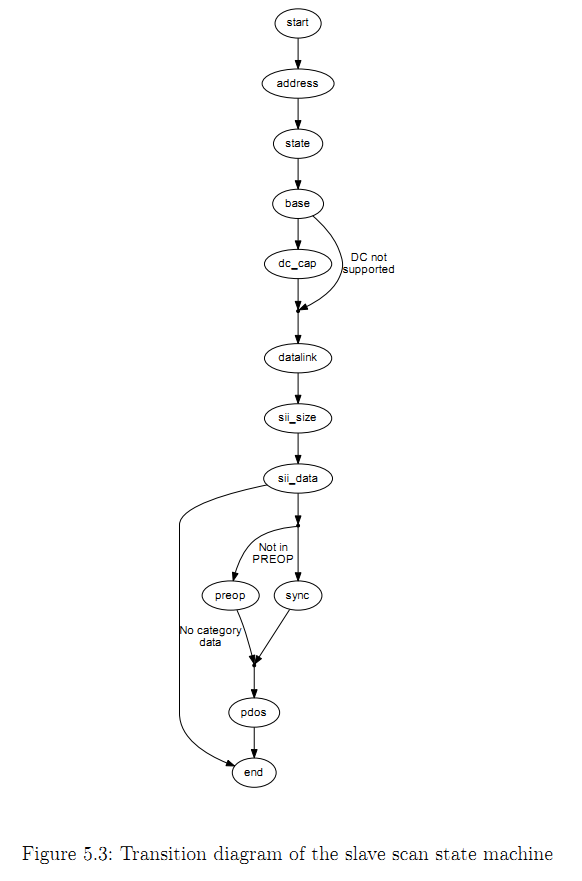
**基本信心** 从底层物理内存中读取基本信息（例如支持的FMMU数量）。

**数据连接** 读取物理端口的信息。

**SII 大小** SII内容的大小决定于分配的SII image内存。

**SII 数据** SII内容被读取状态机image。





**PREOP** 如果从站支持CoE，它被通过使用状态修改FSM设置为PREOP状态，用来使能mailbox通信，并且通过CoE读取PDO配置。

**PD0s** 通过CoE使用PDO读取FSMP 来读取PDOs。如果必要，SII的PDO信息会被重写。

### 从站配置状态机

从站配置状态机，可以在图5.4中看到，引领整个配置从站的过程并且带它进入到一个 特定的应用层状态。

**初始化** 使用状态修改FSM来让从站进入到初始化（INIT）状态。

**清除FMMU** 为了避免从站响应任何数据，清空FMMU。如果从站不支持FMMU，这个状态被跳过。如果请求的是INIT状态，这个状态机就结束了。

**Mailbox Sync Manager Configuration（信箱同步管理配置）** 如果从站支持mailbox通信，配置信箱同步管理器。否则这个状态被跳过。

**PREOP** 使用状态修改FSM将从站带入到PREOP状态。如果请求的是这个状态， 状态机结束。

**SDO 配置** 如果连接到一个从站上，并且应用程序提供了SDO配置，将这些配置发送给从站。

**PDO 配置** 执行PDO配置状态机应用与所有必要的PDO配置工作。

**PDO通过管理器配置** 如果存在PDO同步管理器，配置他们。

**FMMU配置** 如果应用程序提供了FMMU配置（例如，如果应用程序注册PDO入口），应用这些信息。

**SAFEOP** 使用状态改变FSM将从站带入到SAFEOP状态下。如果这是请求的状态，状态机结束。