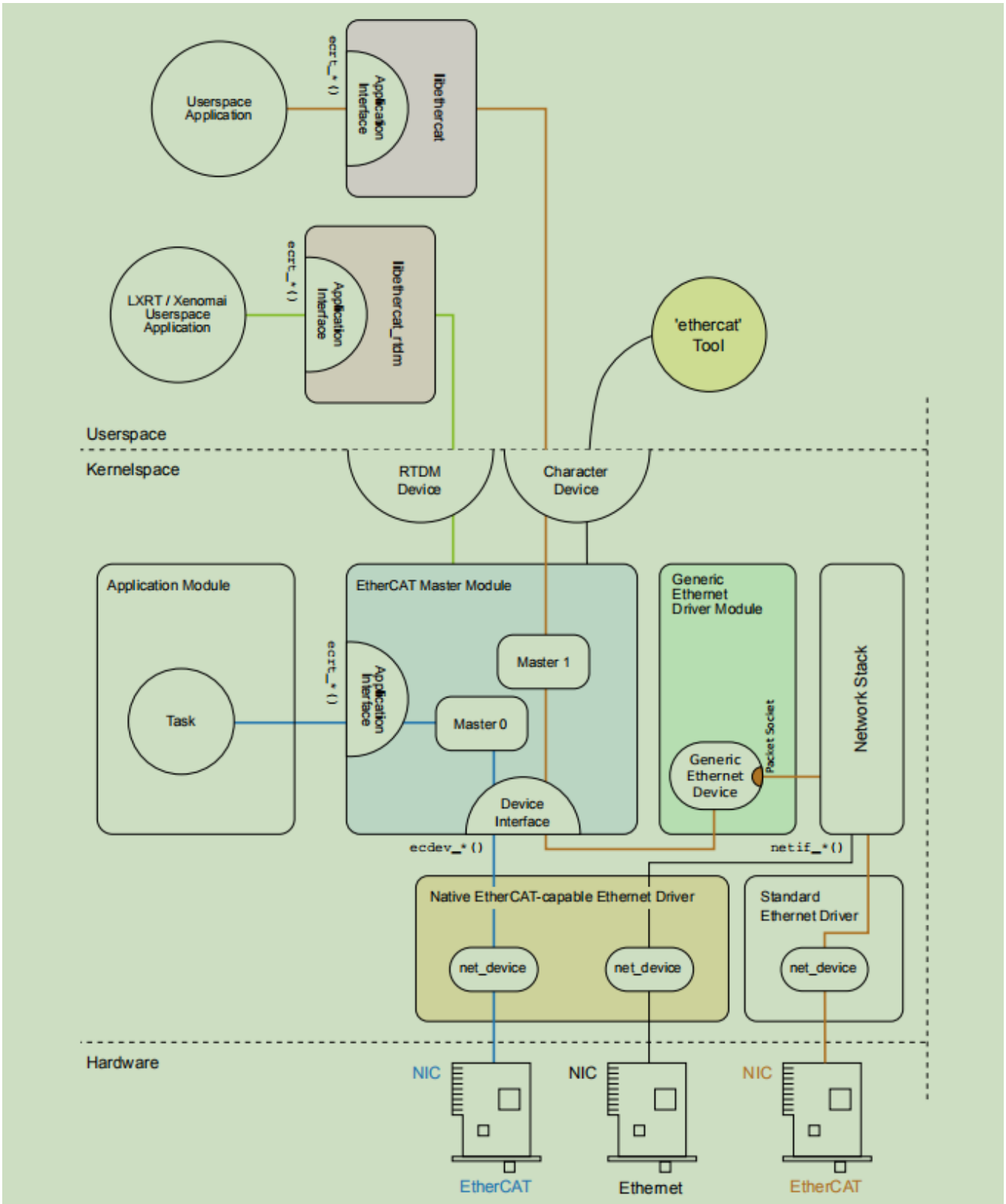
igh架构

igh架构

一、igh主站架构
二、主站内核模块
2.1、加载主站内核模块
2.2、主站运行阶段
三、设备驱动模块
3.1、网络驱动基础
3.2、专用ethercat设备驱动
3.3、通用ethercat设备驱动
四、应用模块
4.1、命令行工具
4.1.1、设备文件接口
4.1.2、设置别名地址
4.1.3、显示总线配置
4.1.4、用c预言输出pdo信息
4.1.5、显示过程数据
4.1.6、配置域
4.1.7、将总线拓扑输出为图形
4.1.8、显示主站和以太网设备信息
4.1.9、列出同步管理器，pdo和pdo条目
4.1.10、输出一个从站的寄存器内容
4.1.11、显示总线上的从站
4.2、用户空间库接口
4.3、RTDM实时接口
4.4、初始化、配置和启动主站

一、igh主站架构

igh主站架构如图所示，其基本通信结构由**硬件层**、**内核空间**及**用户空间**三部分组成。其中图中有颜色的区域都属于igh软件包中的主站模块、库、工具、驱动等。其中主站运行环境主要分为**主站内核模块**、**设备驱动模块**和**应用模块**。



二、主站内核模块

主站内核模块**ec_master**包含一个或多个ethercat主站实例，设备接口和内核空间的应用程序接口。一般一个主站设备只运行一个主站实例。

2.1、加载主站内核模块

主站内核模块可以包含多个主站实例，每个主站实例都会等待一个具有mac地址的以太网设备(不考虑冗余的情况下)。因此要在配置主站环境的时候将主站实例和网卡绑定。

下面的命令加载只有一个主站实例的主站内核模块**ec_master**，该主站实例绑定mac地址为00:0E:0C:DA:A2:20的网卡。

```
modprobe ec_master main_devices=00:0E:0C:DA:A2:20
```

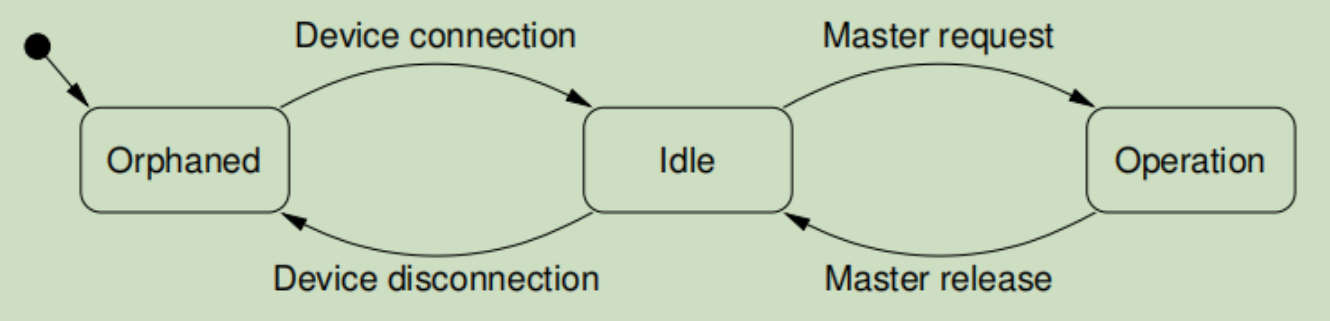
加载具有多个主站实例的主站内核模块**ec_master**如下，每个主站绑定的网卡mac地址用逗号分开。

```
modprobe ec_master main_devices=00:0E:0C:DA:A2:20,00:e0:81:71:d5:1c
```

加载的两个主站实例可以分别通过索引0和1来访问，在加载主站后，当应用程序使用应用程序接口**ecrt_master_request()** 来申请主站时需要传入索引表示申请哪个主站。

2.2、主站运行阶段

每个ethercat主站会经过图中所示几个运行阶段。



孤立阶段 主站内核模块加载后，网络设备连接之前。在这个状态下总线没有数据通信。
空闲阶段 主站已经接受所有必须的以太网设备的连接，但还没有应用程序请求。此时主站从孤立阶段转换到空闲阶段，此时主站运行**主站状态机**，该状态机自动扫描总线来寻找从站，并执行来自用户空间接口的操作。此时可以使用命令行工具访问总线，但是由于缺少总线配置，因此没有过程数据交换。
操作阶段 当应用程序运行时，初始化时期会进行总线配置，请求主站模块，这个时候主站会从空闲阶段转化到操作阶段。在这个阶段主站模块可以向应用程序提供总线上的从站状态信息和过程数据。

三、设备驱动模块

EtherCAT协议基于以太网标准，因此主站依靠标准以太网硬件与总线进行通信。EtherCAT主站下可能会有以下两种以太网驱动模块。
专用以太网驱动模块 专用以太网驱动模块如上图中的黄色部分，它既包含ethercat驱动也包含普通ethernet驱动。这两个驱动可以并行运行。主站内核模块内核模块可以通过**设备接口**直接通过专用以太网驱动模块中的ethercat驱动访问ethercat设备。
通用以太网驱动模块 通用以太网驱动模块如上图中的绿色部分，它负责主站内核模块和协议栈底层部分的连接。这个模块的优点是使得任意的网卡驱动都可以驱动ethercat设备，缺点**这个方法不支持实时接口**，因为这个方法经过了协议栈，因此性能会下降。

3.1、网络驱动基础

网卡驱动的任务 网卡驱动通常负责处理物理层和数据链路层。在发送的时候，它从内核的网络协议栈获取数据然后传给硬件网卡，然后网卡发送出去。在接收的时候，数据先由网卡接收保存在某个缓冲区中(通常是通过dma保存到内存ringbuf中)，然后通过中断通知驱动去读取ringbuf中的数据，现在主流情况下驱动会通过napi机制在收到第一次接收中断后主动进入轮询，满足一定条件再退出。

驱动启动加载 内核在启动时会处理设备树中的网卡设备信息，转化成平台设备注册到平台总线；同时网卡驱动已平台设备驱动的形式也注册到平台总线，然后触发驱动中的probe函数，在这个函数中初始化网卡。启动网卡则在启动服务里面。

中断操作 网络设备通常提供硬件中断，该硬件中断用于分别通知驱动程序收到的帧以及传输成功或发生错误。 驱动程序必须注册一个中断服务程序（ISR），该程序每次在硬件发出此类事件信号时执行。 如果中断是由自己的设备抛出的（多个设备可以共享一个硬件中断），则必须通过读取设备的中断寄存器来确定中断的原因。 例如，如果设置了接收帧的标志，则必须将帧数据从硬件复制到内核内存，然后传递到网络堆栈。

net_device结构体 驱动程序为每个网卡设备注册一个net_device结构体，以与协议栈进行通信并创建**"网络接口"**，对于以太网的接口显示为ethX。net_device结构体通过子结构体net_device_ops中的成员函数处理事件。不是该结构体中的所有成员函数都必须实现，但以下几个函数大多数情况下是需要的：

open() net_device_ops中的ndo_open()成员函数，启动网卡时调用，可以在启动服务中启动或者命令行工具启动如下所示：**ip link set ethX up**，启动网卡后可以接收和发送网络帧。

stop() net_device_ops中的ndo_stop()成员函数，关闭网卡时调用，如**ip link set ethX down**，关闭网卡后就停止接收和发送网络帧。

hard_start_xmit() net_device_ops中的ndo_start_xmit()成员函数，对于必须发送的每个帧都调用此函数，协议栈将帧作为指向sk_buff结构的指针传递。

get_status() net_device_ops中的ndo_get_stats()成员函数，次函数返回设备的net_device_stats结构的指针，这个结构保存该设备的帧的统计信息。每次接收，发送帧或发生错误时，都必须增加此结构中的适当计数器。

设置好net_device结构体后就通过register_netdev()注册。

netif接口 netif接口是协议栈和网卡驱动之间的关卡。例如，成功打开设备后，必须通知网络协议栈，它现在可以将帧传递到网卡驱动，这是通过netif_start_queue()完成的，调用后可以通过hard_start_xmit()发送网络帧。网络驱动中通常会管理着传输队列，如果发送的队列满了则驱动可以调用netif_stop_queue()通知协议栈停止发送网络帧到驱动中。当驱动的发送队列有空位后可以调用netif_wake_queue()来恢复协议栈发送。另一个重要的netif接口是netif_receive_skb()，它将网卡刚接收到的网络帧传给协议栈。

sk_buff sk_buff即套接字缓冲区，它是协议栈和网卡驱动中传输数据的关键数据结构，通过这个结构体的某些成员可以很方便的增加对应层的头部和尾部也很容易剥离头部和尾部。其中**head**指向整个缓冲区的开始，**end**指向整个缓冲区的结尾，**data**指向数据的开始，**tail**指向数据的结尾，head和end在分配了缓冲区后是不变的，而data和tail随着sk_buff在各层之间的流动而改变。通过**skb_push()**在data端增加数据，**skb_put()**在tail端增加数据，**skb_pull()**在data端删除数据，**skb_trim()**在tail端删除数据。

3.2、专用ethercat设备驱动

由于项目没有专用ethercat网卡，因此暂且不用专用ethercat设备驱动，因此暂不分析。

3.3、通用ethercat设备驱动

通用以太网驱动模块ec_generic会扫描被以太网驱动注册进内核协议栈的网卡接口。这个模块提供所有的网卡设备给ethercat主站。如果主站接受了一个设备，那通用以太网驱动模块会创建一个**SOCK_RAW**类型的socket绑定该设备。然后主站模块的设备接口函数就会操作这个socket。

通用驱动模块的优点

- 所有具有linux以太网驱动的以太网设备都可以用来作为ethercat设备。

- 不需要更改以太网驱动（感觉还是要改，见下文）。

  通用驱动模块的缺点

- 性能比使用专用ethercat网卡驱动差，因为要经过协议栈。
- 不能使用实时系统的实时接口，因为。。。

四、应用模块

4.1、命令行工具

4.1.1、设备文件接口

每个主站实例都有一个设备节点文件导出到用户空间，名为/dev/EtherCATx，其中x是主站实例的索引。该设备文件通过通过udev机制自动创建。

4.1.2、设置别名地址

```
ethercat alias [ OPTIONS ] < ALIAS >
```

4.1.3、显示总线配置

```
ethercat config [ OPTIONS ]
```

4.1.4、用c预言输出pdo信息

```
ethercat cstruct [ OPTIONS ]
```

4.1.5、显示过程数据

```
ethercat data [ OPTIONS ]
```

4.1.6、配置域

```
ethercat domains [ OPTIONS ]
```

4.1.7、将总线拓扑输出为图形

```
ethercat graph [ OPTIONS ]
```

4.1.8、显示主站和以太网设备信息

```
ethercat master [ OPTIONS ]
```

4.1.9、列出同步管理器，pdo和pdo条目

```
ethercat pdos [ OPTIONS ]
```

4.1.10、输出一个从站的寄存器内容

```
ethercat reg_read [ OPTIONS ] < ADDRESS > [ SIZE ]
```

4.1.11、显示总线上的从站

```
ethercat slaves [ OPTIONS ]
```

4.2、用户空间库接口

本机应用程序接口位于内核空间中，因此只能从内核内部进行访问。 为了使应用程序接口可从用户空间程序获得，已创建了一个用户空间库，该库可以根据LGPL版本的条款和条件链接到程序。该库名为libethercat。 它的源代码位于lib /子目录中，并且在使用make时默认情况下进行构建。 它以libethercat.a（用于静态链接），libethercat.la（用于libtool）和libethercat.so（用于动态链接）的形式安装在安装前缀下面的lib /路径中。

```
$include <ecrt.h>
int main (void)
{
    ec_master_t * master = ecrt_request_master (0) ;
    if (! master )
    return 1; // error
    pause () ; // wait for signal
    return 0;
}
```

以上示例程序可用以下命令**动态链接**编译：

```
gcc ethercat . c -o ectest -I / opt / etherlab / include \
    -L / opt / etherlab / lib - lethercat \
    -Wl , - - rpath -Wl ,/ opt / etherlab / lib
```

也可以用下面命令**静态链接**编译：

```
gcc - static ectest . c -o ectest -I / opt / etherlab / include \
    / opt / etherlab / lib / libethercat . a
```

4.3、RTDM实时接口

当使用Xenomai或RTAI等实时扩展的用户空间接口时，不建议使用ioctl（ ），因为它可能会干扰实时操作。 为此，已经开发了实时设备模型（RTDM）。 如果主源使用--enable rtdm配置，则主模块除了普通字符设备外，还提供RTDM接口。

要强制应用程序使用RTDM接口而不是普通字符设备，必须将其与libethercat rtdm库而不是libethercat链接。libethercat rtdm的使用是透明的，因此可以照常使用带有完整API的EtherCAT标头ecrt.h。然后链接编译命令更改如下：

```
gcc ethercat - with - rtdm . c -o ectest -I / opt / etherlab / include \
    -L / opt / etherlab / lib - lethercat_rtdm \
    -Wl , - - rpath -Wl ,/ opt / etherlab / lib
```

4.4、初始化、配置和启动主站

初始化脚本 etc/init.d/ethercat

配置脚本 etc/sysconfig/ethercat，如果使用systemd则为/etc/ethercat.conf

启动主站 可使用systemd自动启动，或者手工etc/init.d/ethercat restart启动