**AN1811**
**应用笔记**

# UM324xF EMAC 配置指南

版本：V1.0




广芯微电子（广州）股份有限公司

http://www.unicmicro.com/

# 条款协议



# 目录





# 1 摘要

本篇应用笔记主要介绍UM324xF EMAC配置指南。

本篇应用笔记主要包括：

- EMAC概述
- EMAC配置流程

注：具体功能及寄存器的操作等相关事项请以用户手册为准。

# 2 EMAC 概述

UM324xF EMAC 可以接收和发送以太网数据，符合 IEEE802.3-2002 标准。

## 2.1 特性

- 支持 MII、RMII 和 RGMII 接口
- 支持外部 PHY 接口实现 10M/100M/1000M bit/s 的数据传输速率
- 支持全双工和半双工工作模式
- 使用 MDIO 接口配置和管理 PHY 设备
- 支持以太网时间戳（IEEE 1588-2002）。每个帧的发送或接收状态下给出 64 位时间戳
- 报头和帧起始数据(SFD)在发送路径中插入、在接收路径中删除
- 支持帧长有效性检测，丢弃超长帧和超短帧。
- 支持对输入帧进行 CRC 校验，可丢弃校验错的帧。
- 支持对输出帧添加 CRC 校验。
- 支持短帧填充功能。
- 支持对接收和发送帧进行统计计数。
- 支持广播帧、组播帧和单播帧过滤。
- 支持控制报文、IP 报文、广播或多播报文限速处理功能可配置。
- 支持包过滤功能。
- 支持入队中断和超时中断两种中断方式。
- 支持收发包缓存。
- 支持 EEE 能效以太网功能。
- 相互独立的两个 2K 字节的 FIFO 分别用于发送与接收



## 2.2 管脚说明

| 功能管脚 | RGMII | RMII | MII | 复用管脚 | 方向 | 功能描述 |
|---|---|---|---|---|---|---|
| ETH_GTXCLK/ ETH_TXCLK | ETH_GTXCLK | - | ETH_TXCLK | PC3 | Input/ Output | RGMII：通道发送时钟 125M 输出<br>MII：通道发送时钟输入 |
| ETH_GTXCLK/ ETH_TXCLK | ETH_RXCLK | ETH_RXCLK/ ETH_REF_50M | ETH_RXCLK | PA1 | Input | 通道接收时钟 10M/100M/1000M，RGMII/RMII/MII, RMII(50M) |
| ETH_CLK125M (REF_50M_125M) | ETH_CLK125M | - | - | PD2 | input | 通道参考 125M 时钟，RGMII |
| ETH_TXD0 | ETH_TXD0 | ETH_TXD0 | ETH_TXD0 | PB12 | Output | 通道发送数据 0 |
| ETH_TXD1 | ETH_TXD1 | ETH_TXD1 | ETH_TXD1 | PB13 | Output | 通道发送数据 1 |
| ETH_TXD2 | ETH_TXD2 | - | ETH_TXD2 | PC2 | Output | 通道发送数据 2 |
| ETH_TXD3 | ETH_TXD3 | - | ETH_TXD3 | PB8 | Output | 通道发送数据 3 |
| ETH_TXEN | ETH_TXEN | ETH_TXEN | ETH_TXEN | PB11 | Output | 通道发送数据使能 |
| ETH_RXD0 | ETH_RXD0 | ETH_RXD0 | ETH_RXD0 | PC4 | input | 通道接收数据 0 |
| ETH_RXD1 | ETH_RXD1 | ETH_RXD1 | ETH_RXD1 | PC5 | Input | 通道接收数据 1 |
| ETH_RXD2 | ETH_RXD2 | - | ETH_RXD2 | PB0 | Input | 通道接收数据 2 |
| ETH_RXD3 | ETH_RXD3 | - | ETH_RXD3 | PB1 | Input | 通道接收数据 3 |
| ETH_RXDV | ETH_RXDV | ETH_RXDV | ETH_RXDV | PA7 | Input | 通道接收数据有效 |
| ETH_MDIO | ETH_MDIO | ETH_MDIO | ETH_MDIO | PA2 | Input/ Output | 通道串行数据 |



| 功能管脚 | RGMII | RMII | MII | 复用管脚 | 方向 | 功能描述 |
|---|---|---|---|---|---|---|
| ETH_MDC | ETH_MDC | ETH_MDC | ETH_MDC | PC1 | Output | 通道串行时钟输出 |

## 2.3 MII&RMII&GMII 接口介绍

### 2.3.1 概述

MII(media independent interface 介质无关接口或媒体独立接口)，它是 IEEE-802.3 定义的以太网行业标准。它包括一个数据接口和一个 MAC 和 PHY 之间的管理接口。

数据接口包括分别用于发送器和接收器的两条独立信道，每条信道都有自己的数据、时钟、控制信号。MII 数据接口总共需要 16 个信号。

管理接口是个双信号接口：一个是时钟信号，另一个是数据信号，通过管理接口，上层能监控和控制 PHY。MII 只有两条信号线。

MII 标准接口用于连接 Fast Ethernet Mac Block 与 PHY。表明在不对 MAC 硬件重新设计或替换的情况下，如何类型的 PHY 设备都可以正常工作。在其他速率下工作的与 MII 等效的接口有：AUI(10M 以太网)，GMII(Gigabit 以太网)和 XAUI(10Gigabit 以太网)。

### 2.3.2 MII 总线

在 IEEE802.3 中规定的 MII 总线是一种用于将不同类型的 PHY 与相同网络控制器（MAC）相连接的通用总线。网络控制器可以用同样的硬件接口与任何 PHY 进行连接。

**MII 相关接口介绍：**

以太网媒体接口有：MII&RMII&SMII&GMII

所有的这些接口都从 MII 而来，MII 是（Medium Independent Interface）的意思，是指不用考虑媒体是铜轴、光纤、电缆等，因为这些媒体处理的相关工作都有 PHY 或者叫做 MAC 的芯片完成。

MII 支持 10 兆和 100 兆的操作，一个接口由 14 根线组成，它的支持还是比较灵活的，但是有一个缺点是因为它一个端口用的信号线太多，如果一个 8 端口的交换机要用到 112 根线，16 端口就要用到 224 根线，到 32 端口的话就要用到 448 根线，一般按照这个接口做交换机，是不太现实的，所以现代的交换机的制作都会用到其它的一些从 MII 简化出来的标准，比如 RMII、SMII、GMII 等。

RMII 是简化的 MII 接口，在数据的收发上它比 MII 接口少了一倍的信号线，所以它一般要求是 50 兆的总线时钟。RMII 一般用在多端口的交换机，它不是每个端口安排收、发两个时钟，而是所有的数据端口公用一个时钟用于所有端口的收发，这里就节省了不少的端口数目。RMII 的一个端口要求 7 个数据线，比 MII 少了一倍，所以交换机能够接入多一倍数据的端口。和 MII 一样，RMII 支持 10 兆和 100 兆的总线接口速度。

SMII 是由思科提出的一种媒体接口，它有比 RMII 更少的信号线数目，S 表示串行的意思。因为它只用一根信号线传送发送数据，一根信号线传输接受数据，所以在时钟上为了满足 100 的需求，它的时钟频率很高，达到了 125 兆，为什么用 125 兆，是因为数据线里面会传送一些控制信息。SMII 一个端口仅用 4 根信号线完成 100 信号的传输，比起 RMII 差不多又少了一倍的信号线。SMII 在工业界的支持力度是很高的。同理，所有端口的数据收发都公用同一个外部的 125M 时钟。

GMII 是千兆网的 MII 接口，这个也有相应的 RGMII 接口，表示简化了的 GMII 接口。

### 2.3.3 MII 工作原理

“媒体独立”表明在不对 MAC 硬件重新设计或替换的情况下，任何类型的 PHY 设备都可以正常工作。包括分别用于发送器和接收器的两条独立信道。每条信道都有自己的数据、时钟和控制信号。

MII 数据接口总共需要 16 个信号，包括 TX_ER, TXD, TX_EN, TX_CLK, COL, RXD, RX_EX, RX_CLK，CRS，RX_DV 等。

MII 以 4 位半字节方式传送数据双向传输，时钟速率 25MHz。其工作速率可达 100Mb/S。

MII 管理接口是个双信号接口，一个是时钟信号，另一个是数据信号。

通过管理接口，上层能监视和控制 PHY，其管理是使用 SMI（Serial Management Interface）总线通过读写 PHY 的寄存器来完成的。

PHY 里面的部分寄存器是 IEEE 定义的，这样 PHY 把自己的目前的状态反映到寄存器里面，MAC 通过 SMI 总线不断的读取 PHY 的状态寄存器以得知目前 PHY 的状态，例如连接速度，双工的能力等。

当然也可以通过 SMI 设置 PHY 的寄存器达到控制的目的，例如流控的打开关闭，自协商模式还是强制模式等。

不论是物理连接的 MII 总线和 SMI 总线还是 PHY 的状态寄存器和控制寄存器都是有 IEEE 的规范的，因此不同公司的 MAC 和 PHY 一样可以协调工作。当然为了配合不同公司的 PHY 的自己特有的一些功能，驱动需要做相应的修改。

PHY 是物理接口收发器，它实现物理层。包括 MII/GMII（介质独立接口）子层、PCS（物理编码子层）、PMA（物理介质附加）子层、PMD（物理介质相关）子层、MDI 子层。100BaseTX 采用 4B/5B 编码。

PHY 在发送数据的时候，收到 MAC 过来的数据（对 PHY 来说，没有帧的概念，对它来说，都是数据而不管什么地址，数据还是 CRC），每 4bit 就增加 1bit 的检错码，然后把并行数据转化为串行流数据，再按照物理层的编码规则把数据编码，再变为模拟信号把数据送出去。收数据时的流程反之。

PHY 还有个重要的功能就是实现 CSMA/CD 的部分功能。

它可以检测到网络上是否有数据在传送，如果有数据在传送中就等待，一旦检测到网络空闲，再等待一个随机时间后将送数据出去。如果两个碰巧同时送出了数据，那样必将造成冲突，这时候，

冲突检测机构可以检测到冲突，然后各等待一个随机的时间重新发送数据。这个随机时间很有讲究的，并不是一个常数，在不同的时刻计算出来的随机时间都是不同的，而且有多重算法来应付出现概率很低的同两台主机之间的第二次冲突。

通信速率通过双方协商，协商的结果是两个设备中能同时支持的最大速度和最好的双工模式，这个技术被称为 Auto Negotiation 或者 NWAY。

隔离变压器把 PHY 送出来的差分信号用差模耦合的线圈耦合滤波以增强信号，并且通过电磁场的转换耦合到连接网线的另外一端。

RJ-45 中 1、2 是传送数据的，3、6 是接收数据的。

新的 PHY 支持 AUTO MDI-X 功能，也需要隔离变压器支持，它可以实现 RJ-45 接口的 1、2 上的传送信号线和 3、6 上的接收信号线的功能自动互相交换。

### 2.3.4 GMII 接口简介

GMII（Gigabit MII），GMII 采用 8 位接口数据，工作时钟 125MHz，因此传输速率可达 1000Mbps。同时兼容 MII 所规定的 10/100 Mbps 工作方式。

- GMII 接口数据结构符合 IEEE 以太网标准。该接口定义见 IEEE 802.3-2000。
- 发送器：
  - GTXCLK——吉比特 TX.。信号的时钟信号（125MHz）
  - TXCLK——10/100M 信号时钟
  - TXD［7:0］——被发送数据
  - TXEN——发送器使能信号
  - TXER——发送器错误（用于破坏一个数据包）

注：在千兆速率下，向 PHY 提供 GTXCLK 信号，TXD、TXEN、TXER 信号与此时钟信号同步。否则，在 10/100M 速率下，PHY 提供 TXCLK 时钟信号，其它信号与此信号同步。其工作频率为 25MHz（100M 网络）或 2.5MHz（10M 网络）。

- 接收器：
  - RXCLK——接收时钟信号（从收到的数据中提取，因此与 GTXCLK 无关联）
  - RXD［7:0］——接收数据
  - RXDV——接收数据有效指示
  - RXER——接收数据出错指示
  - COL——冲突检测（仅用于半双工状态）
- 管理配置：
  - MDC——配置接口时钟
  - MDIO——配置接口 I/O

- 管理配置接口控制 PHY 的特性。该接口有 32 个寄存器地址，每个地址 16 位。其中前 16 个已经在“IEEE 802.3，2000-22.2.4 Management Functions”中规定了用途，其余的则由各器件自己指定。

### 2.3.5 RMII 简介

RMII： Reduced Media Independant Interface 即简化媒体独立接口；是标准的以太网接口之一，比 MII 有更少的 I/O 传输。

**关于 RMII 口和 MII 口的问题。**

- RMII 口是用两根线来传输数据的。
- MII 口是用 4 根线来传输数据的。
- GMII 是用 8 根线来传输数据的。
- MII/RMII 只是一种接口，对于 10M 线速，MII 的速率是 2.5M，RMII 则是 5M；对于 100M 线速，MII 的速率是 25M，RMII 则是 50M。
- MII/RMII 用于传输以太网包，在 MII/RMII 接口是 4/2bit 的，在以太网的 PHY 里需要做串并转换、编解码等才能在双绞线和光纤上进行传输，其帧格式遵循 IEEE 802.3（10M）/IEEE 802.3u（100M）/IEEE 802.1q（VLAN）。
- 以太网帧的格式为：前导符+开始位+目的 Mac 地址+源 Mac 地址+类型/长度+数据+Padding（Optional）+32bitCRC。
- 如果有 Vlan，则要在类型/长度后面加上 2 个字节的 Vlan Tag，其中 12bit 来表示 Vlan Id，另外 4bit 表示数据的优先级。

## 2.4 以太网 DMA

### 2.4.1 什么是以太网 DMA

DMA 就是不需要 CPU 的参与就能够实现内存和外设之间的数据交换，同样的，以太网 DMA 的作用也是如此，它的作用就是在不需要 CPU 的参与下，实现内存和以太网外设的数据交换。

用通俗一点的话来表述，就是我们将要发送的数据放到一片内存去，告诉以太网 DMA，我已经将数据放过去了，你去取出来发送到网络中去吧。当网络数据来了的时候，以太网 DMA 自动将数据拷贝到一片内存中，产生中断告诉 CPU，数据来了，你去取出来吧。



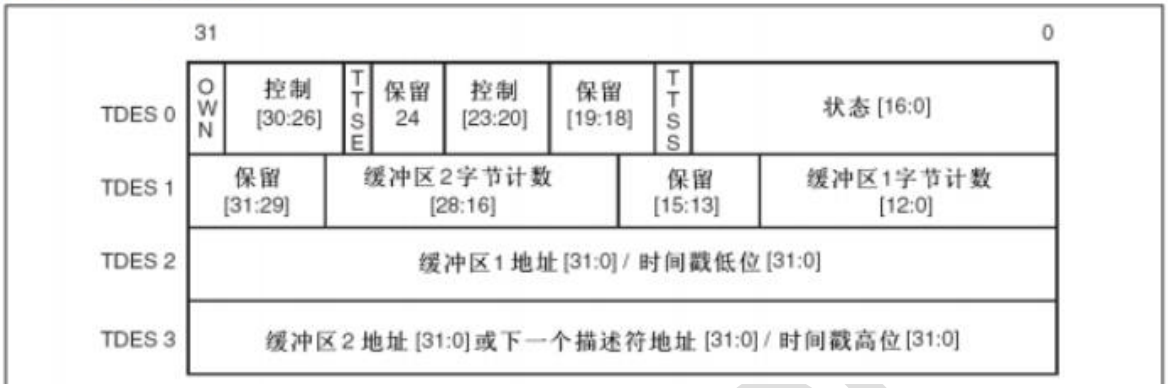
  

## 2.4.2 DMA 描述符的本质是什么

**发送DMA描述符**

描述符结构体包含4个32位字，TDES0、TDES1、TDES2和TDES3的位定义如下图：

图325 传输描述符




一眼看上去好像很复杂，这些是寄存器吗？但是找了一大堆文档，也没有找到它的寄存器地址，因为它根本就不是什么寄存器，而是 4 个 32Bit 的内存。对的，你去找发送描述符的硬件结构，是肯定找不到的，因为它完全是纯软件的概念，它的本质就是我们自己用结构体来实现这个描述符，然后将描述符的首地址写入到【ETH_DMATDLAR】寄存器中，CPU 就知道这片内存是用来作为发送描述符了。

发送描述符的主要作用就是用来记录发送缓冲区的大小，缓冲区的地址，还有这个缓冲区的状态等等，里面有很多信息，这些信息是用来协同 CPU 和 DMA 二者之间工作的，我把他们的功能简要的写了出来，如下所示：

TDES0/RDES0 主要用来表示描述符的状态和控制信息。

TDES1/RDES1 表示该描述符缓冲区数据的有效长度。

TDES2/RDES2 表示描述符缓冲区的地址，我们要发送的数据，就是放在这个地址所指向的内存中。

TDES3/RDES3 表示下一个描述符的地址。

我们通过定义一个结构体来实现这个发送描述符，如下所示：

```
typedef struct  {
  uint32_t    Status;                 /*!< Status */
  uint32_t    ControlBufferSize;      /*!< Control and Buffer1, Buffer2 lengths */
  uint32_t    Buffer1Addr;            /*!< Buffer1 address pointer */
  uint32_t    Buffer2NextDescAddr;    /*!< Buffer2 or next descriptor address pointer */
} ETH_DMADESCTypeDef;
```

当我们需要发送数据的时候，我们把数据拷贝到发送描述符的缓冲区中（Buffer1Addr），告诉 DMA 我们拷贝完成了，DMA 就会从发送描述符的缓冲区中取数据，将数据通过以太网外设发送到网络中去。

同样地，以太网外设接收到了网络中的数据时，DMA 自动拷贝数据到接收描述符的缓冲区中（Buffer1Addr），产生中断告诉 CPU，有数据来了，我们就可以取出描述符的数据，从而知道接收到了什么。

### 2.4.3 DMA 和 CPU 如何有序访问描述符

描述符的本质就是 4 个 32 位的内存，描述符的 Status 表示该描述符的状态和控制信息。因为描述符是 DMA 和 CPU 二者之间的共享内存，既然是共享资源，就需要进行保护，当 DMA 在使用的时候，CPU 就不能使用，当 CPU 在使用的时候，DMA 就不能使用。这个使用权的控制，就通过 TDES0 寄存器中的 OWN 位体现出来，对应到发送描述符结构体就是 Status 变量的最高位。

| | |
|---|---|
| 位31 | **OWN**: 占有位 (Own bit)<br>1：表示DMA占有描述符。<br>0：表示CPU占有描述符。<br>DMA在传输完整个帧或者这个缓存里的数据全部读出以后把该位清'0'。每个帧的第一个缓存描述符的占有位，必须在后面缓存描述符的占有位全部置'1'以后，才能置'1'。 |

我们看上图可知，当 OWN 位为 0 的时候，表示 CPU 可以将要发送的数据拷贝到描述符中，拷贝完成以后，我们手动将描述符的 OWN 位设置为 1，以此来告诉 DMA 控制器，我已经拷贝完数据了，你可以从描述符中取出数据进行发送了。这时候 DMA 就会取出描述符中的数据，将数据发送出去，DMA 在操作完描述符以后，自动将 OWN 位设置为 0，告诉 CPU，我 DMA 已经发送完数据啦，你可以拷贝下一帧数据到描述符上了。整个发送的过程就是这样配合的。

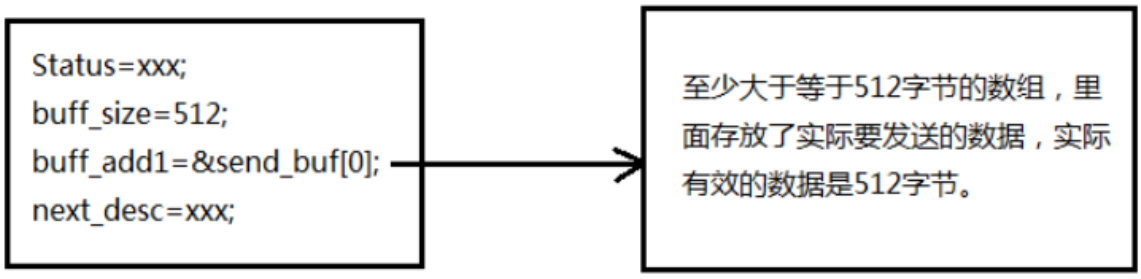
### 2.4.4 发送的数据实际放在哪里

发送 DMA 描述符只有 4 个 32Bit 的内存空间，这点内存肯定放不下我们要发送的以太网数据帧，那么我们要发送的数据实际上是存在描述符的哪里呢？

| | | | | |
|---|---|---|---|---|
| TDES 1 | 保留<br>[31:29] | 缓冲区2字节计数<br>[28:16] | 保留<br>[15:13] | 缓冲区1字节计数<br>[12:0] |
| TDES 2 | 缓冲区1地址 [31:0] / 时间戳低位 [31:0] | | | |

TDES2 就是用来设置存放要发送数据缓冲区的首地址的，TDES1 是用来告诉 DMA 这个缓冲区有效数据长度的。我们可以开辟一个数组，将数组的首地址写入 Buffer1Addr 中，这样就设置好了发送描述符实际存放数据的内存地址，我们往这个数组中写入要发送的数据，再把数据长度写入 ControlBufferSize 中，DMA 就可以从这片内存中取 ControlBufferSize 长度的数据出来进行发送了。

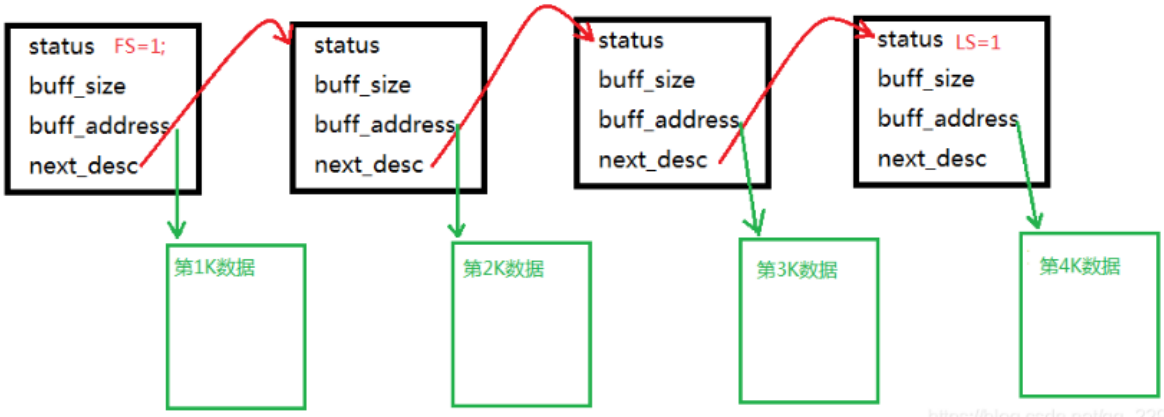
例如，我们要发送一帧 512 字节的数据，那么我们就需要先建立一个至少大于 512 字节的数组，将要发送的数据拷贝到这个数组里面，然后设置这个描述符的数据长度是 512 即可，如下图所示。




### 2.4.5 要发送的数据长度超过一个描述符能够存放的最大长度怎么办

如果我们发送的一帧数据很大，一个描述符没有办法放下那么多数据，应该怎么办呢？这时候就需要用到链表，将这帧以太网数据分割位若干部分，分别存放在多个描述符里面，描述符之间用链表的形式建立连接。说起来有点抽象，我们举个例子。

例如，有 4K 字节的一帧数据要发送出去，但是每一个描述符的缓冲区大小只有 1K，这时候就需要用 4 个描述符来存储要发送的这一帧数据，请看下图。我们把第一个 1K 的数据放入描述符中，并且设置它的 TEDS0 寄存器的 FS 位为 1，表示这个描述符存储了数据帧的第一个分块，把最后 1K 的数据放入描述符中，设置它的 TDES0 寄存器的 LS 位为 1，表示这是该帧数据的最后一个分块。这样 DMA 就能够根据这些信息组合出一条完整的数据帧，进行发送。




# 3 EMAC 配置流程

## 3.1 GPIO 初始化

确定使用的 PHY 的芯片接口，根据不同的接口做不同的配置

### 3.1.1 RMII 接口 GPIO 初始化

1. 使能接口使用的 GPIO 时钟。
2. 如果外部 PHY 的供电由 MCU 控制，先 PHY 上电。
3. 配置 GPIO 的复用功能。

   具体配置如下管脚：RXD0, RXD1, CRS_DV, MDC, MDIO, REF_CLK, TXD0, TXD1, TX_EN

### 3.1.2 RGMII 接口 GPIO 初始化

1. 使能接口使用的 GPIO 时钟。
2. 如果外部 PHY 的供电由 MCU 控制，先 PHY 上电。
3. 配置 GPIO 的复用功能。

   具体配置如下管脚：ETH_TX_CLK，ETH_TX_ER，ETH_TX_EN，ETH_TXD0，ETH_TXD1，ETH_TXD2, ETH_TXD3, ETH_RX_CLK, ETH_RX_DV，ETH_RX_ER，ETH_RXD0，ETH_RXD1，ETH_RXD2，ETH_RXD3，ETH_MDC，ETH_MDIO，ETH_CLK125M_IN，ETH_CRS，ETH_COL

## 3.2 EMAC 时钟使能

通过配置 RCM->AHB0CKENR 寄存器的 BIT10,使能 EMAC 模块

## 3.3 EMAC 接口模式配置

通过配置 SCU->EMACMR 寄存器的最后两个 BIT，可以配置 EMAC 为 MII，RGMII，RMII 模式。

## 3.4 SMI 接口配置

### 3.4.1 SMI 时钟配置

根据系统时钟自动配置 SMI 通信时的时钟，写入寄存器 EMAC->GMIIADDRESS

### 3.4.2 SMI 自动协商 PHY 接口速率

通过 SMI 接口，把自动协商好的接口速率写入 EMAC->CONFIG 配置寄存器。



## 3.5 MAC 寄存器配置

写入 mac 地址，配置前言长度，帧间隔，是否过滤等常规需要的寄存器。

## 3.6 DMA 描述符配置

配置 DMA 接收和发送描述符的地址，大小，并把地址写入 EMAC->DMA_RLA，EMAC->DMA_TLA 寄存器。

## 3.7 使能 EMAC 和 DMA 的接收和发送

1. 配置 EMAC->CONFIG，使能 EMAC 接收和发送。
2. 配置 EMAC->DMA_OPMODE，使能 DMA 的接收和发送。

# 4 版本修订

| 版本 | 日期 | 描述 |
| --- | --- | --- |
| V1.0 | 2023.04.04 | 初始版 |
| | | |
| | | |
| | | |
| | | |