LPC1100 系列微控制器

第十四章 C_CAN 片上驱动

用户手册 Rev.1.00

广州周立功单片机发展有限公司

地址：广州市天河北路 689 号光大银行大厦 12 楼 F4

网址：http://www.zlgmcu.com

销售与服务网络（一）

广州周立功单片机发展有限公司

地址：广州市天河北路 689 号光大银行大厦 12 楼 F4
邮编：510630
电话：(020)38730916　38730917　38730972　38730976　38730977
传真：(020)38730925
网址：www.zlgmcu.com

广州专卖店

地址：广州市天河区新赛格电子城 203-204 室
电话：(020)87578634　87569917
传真：(020)87578842

北京周立功

地址：北京市海淀区知春路 113 号银网中心 A 座 1207-1208 室（中发电子市场斜对面）
电话：(010)62536178　62536179　82628073
传真：(010)82614433

杭州周立功

地址：杭州市天目山路 217 号江南电子大厦 502 室
电话：(0571)89719480　89719481　89719482
89719483　89719484　89719485
传真：(0571)89719494

深圳周立功

地址：深圳市深南中路 2070 号电子科技大厦 C 座 4 楼 D 室
电话：(0755)83781788（5 线）
传真：(0755)83793285

上海周立功

地址：上海市北京东路 668 号科技京城东座 7E 室
电话：(021)53083452　53083453　53083496
传真：(021)53083491

南京周立功

地址：南京市珠江路 280 号珠江大厦 1501 室
电话：(025) 68123901　68123902
传真：(025) 68123900

重庆周立功

地址：重庆市石桥铺科园一路二号大西洋国际大厦（赛格电子市场）1611 室
电话：(023)68796438　68796439
传真：(023)68796439

成都周立功

地址：成都市一环路南二段 1 号数码科技大厦 403 室
电话：(028)85439836　85437446
传真：(028)85437896

武汉周立功

地址：武汉市洪山区广埠屯珞瑜路 158 号 12128 室（华中电脑数码市场）
电话：(027)87168497　87168297　87168397
传真：(027)87163755

西安办事处

地址：西安市长安北路 54 号太平洋大厦 1201 室
电话：(029)87881296　83063000　87881295
传真：(029)87880865

销售与服务网络（二）

广州致远电子有限公司

地址：广州市天河区车陂路黄洲工业区 3 栋 2 楼
邮编：510660
传真：(020)38601859
网址：www.embedtools.com （嵌入式系统事业部）
www.embedcontrol.com （工控网络事业部）
www.ecardsys.com （楼宇自动化事业部）

技术支持：

CAN-bus：

电话：(020)22644381 22644382 22644253
邮箱：can.support@embedcontrol.com

iCAN 及数据采集：

电话：(020)28872344 22644373
邮箱：ican@embedcontrol.com

MiniARM：

电话：(020)28872684 28267813
邮箱：miniarm.support@embedtools.com

以太网：

电话：(020)22644380 22644385
邮箱：ethernet.support@embedcontrol.com

无线通讯：

电话：(020) 22644386
邮箱：wireless@embedcontrol.com

串行通讯：

电话：(020)28267800 22644385
邮箱：serial@embedcontrol.com

编程器：

电话：(020)22644371
邮箱：programmer@embedtools.com

分析仪器：

电话：(020)22644375
邮箱：tools@embedtools.com

ARM 嵌入式系统：

电话：(020) 22644383 22644384
邮箱：NXPARM@zlgmcu.com

楼宇自动化：

电话：(020)22644376 22644389 28267806
邮箱：mjs.support@ecardsys.com
mifare.support@zlgmcu.com

销售：

电话：(020)22644249 22644399 22644372 22644261 28872524
28872342 28872349 28872569 28872573 38601786

维修：

电话：(020)22644245

目 录

第14章 C_CAN 片上驱动

14.1 本章导读

C_CAN 模块只在 LPC11Cxx（LPC11C00 系列）器件上才具有。

14.2 特性

片上驱动程序存放在引导 ROM 中，并通过定义好的 API 向用户应用程序提供 CAN 和 CANopen 初始化和通信特性。下列函数包含在 API 中：

- CAN 设置和初始化；
- CAN 发送和接收报文；
- CAN 状态；
- CANopen 对象字典；
- CANopen SDO 加速通信；
- CANopen SDO 分段通信原语（primitives）；
- CANopen SDO 返回处理。

14.3 概述

除了 CAN ISP 之外，引导 ROM 还提供 CAN 和 CANopen API 来简化 CAN 应用开发程序。它包括初始化、配置、基本 CAN 发送/接收和 CANopen SDO 接口。可以使用回调函数来处理接收事件。

14.4 API 描述

14.4.1 调用 C_CAN API

ROM 中有一个固定的单元包含着一个指向 ROM 驱动程序表的指针，即 0x1FFF 1FF8。该单元对所有的 LPC11C1x 系列 Cortex-M0 微控制器器件是相同的。ROM 驱动程序表包含一个指向 CAN API 表的指针。各种 CAN API 函数的指针存放在该表中。可以使用 C 结构体来调用 CAN API 函数。

图 14.1 解说了用于访问片上 CAN API 的指针机制。CAN API 使用[START_CANAPI_RAM]至[END_API_RAM]地址上的片上 RAM。应用程序不应使用该范围的地址。对于使用片上 CAN API 的应用程序，应适当修改连接器控制文件，以防止将该区域用于应用程序的各种存放操作。

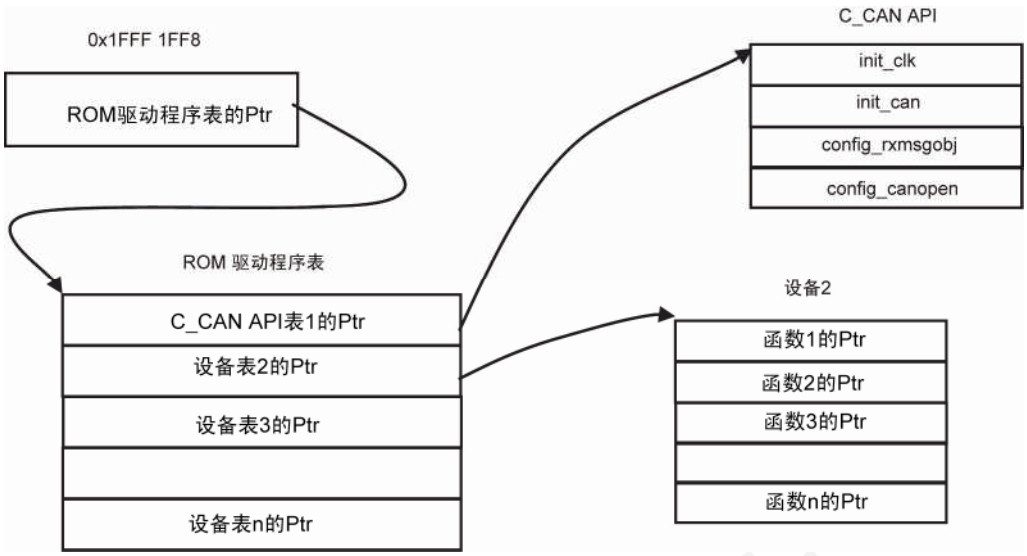

图 14.1 CAN API 指针结构体

用户使用带函数列表的 C 结构体调用 API 函数，结构为如下所示：

```
typedef struct _CAND {
void (*init_can) (uint32_t * can_cfg);
void (*isr) (void);
void (*config_rxmsgobj) (CAN_MSG_OBJ * msg_obj);
uint8_t (*can_receive) (CAN_MSG_OBJ * msg_obj);
void (*can_transmit) (CAN_MSG_OBJ * msg_obj);
void (*config_canopen) (CAN_CANOPENCFG * canopen_cfg);
void (*canopen_handler) (void);
void (*config_calb) (CAN_CALLBACKS * callback_cfg);
} CAND;
```

14.4.2 CAN 初始化

CAN 控制器时钟分频器的 CAN 位速率被置位，CAN 控制器的初始化是在基于寄存器的阵列值上实现的，这些值是通过指针来进行传递。

```
void init_can (uint32_t * can_cfg)
```

阵列中的首个 32 位值被应用到 CANCLKKDIV 寄存器，第二个值则被应用到 CAN_BTR 寄存器。

调用范例：

```
ROM **rom = (ROM **)0x1fff1ff8;
uint32_t CanApiClkInitTable[2] = {
0x00000000UL,                    // CANCLKDIV
0x00004DC5UL                     // CAN_BTR
};
(*rom)->pCANAPI->init_can(&CanApiCanInitTable[0]);
```

14.4.3 CAN 中断处理程序

当用户应用程序有效时，中断处理程序会映射在用户 Flash 空间中。用户应用程序必须为 CAN 中断提供中断处理程序。为了能处理 CAN 事件和调用回调函数，应用程序必须直接从中断处理程序服务中调用 CAN API 中断处理程序。CAN API 中断处理程序根据 CAN 总线上接收到的数据和检测到的状态来采取相关的操作。

```
void isr (void)
```

CAN 中断处理程序不可处理 CANopen 报文。

调用范例：

```
(*rom)->pCAND->isr();
```

14.4.4 CAN Rx 报文对象配置

CAN API 支持和使用具有 32 个报文对象的完整 CAN 模块。任意报文对象都可以用于 11 位或 29 位报文的接收或发送。它还支持 RTR 位（远程发送）置位的 CAN 报文。对于报收对象，报文标识符的屏蔽模式可以允许接收的报文范围，可高至接收单个报文对象里总线上的所有 CAN 报文。

发送报文对象在使用时被自动配置：

```
//CAN_MSG_OBJ.mode_id 的控制位
#define CAN_MSGOBJ_STD 0x00000000UL        //CAN 2.0a 11 位 ID
#define CAN_MSGOBJ_EXT 0x20000000UL        //CAN 2.0b 29 位 ID
#define CAN_MSGOBJ_DAT 0x00000000UL        //数据帧
#define CAN_MSGOBJ_RTR 0x40000000UL        //rtr 帧
typedef struct _CAN_MSG_OBJ {
  uint32_t mode_id;
  uint32_t mask;
  uint8_t data[8];
  uint8_t dlc;
  uint8_t msgobj;
} CAN_MSG_OBJ;
void config_rxmsgobj (CAN_MSG_OBJ * msg_obj)
```

调用范例：

```
//配置报文对象 1 来接收所有 11 位报文 0x000-0x00F
msg_obj.msgobj      = 1;
msg_obj.mask        = 0x7F0;
(*rom)->pCAND-> config_rxmsgobj(&msg_obj);
```

14.4.5 CAN 接收

CAN 接收函数允许读取 Rx 报文对象接收到的报文。指向报文对象结构的指针会被传递到接收函数中。在调用之前，必须要在结构体中设置要被读取的报文对象的编号。

```
void config_rxmsgobj (CAN_MSG_OBJ * msg_obj)
```

调用范例：

```
// 读出接收到的报文
msg_obj.msgobj = 5;
(*rom)->pCAND->can_receive(&msg_obj);
```

14.4.6 CAN 发送

CAN 发送函数允许设置报文对象，并可在总线上触发 CAN 报文的传送。11 位标准和 29 位扩展报文对象被看作为标准数据和远程发送（RTR）报文。

```
void config_txmsgobj (CAN_MSG_OBJ * msg_obj)
```

调用范例：

```
msg_obj.msgobj      = 3;
msg_obj.mode_id     = 0x123UL;
msg_obj.mask        = 0x0UL;
msg_obj.dlc         = 1;
msg_obj.data[0]     = 0x00;
(*rom)->pCAND->can_transmit(&msg_obj);
```

14.4.7 CANopen 配置

CAN API 支持对象字典接口和 SDO 协议。为了激活它，必须要将指针指向带有 CANopen Node ID（1…127）、用于接收和发送 SDO 的报文对象编号和二个指向报文字典配置表和它们大小的指针的结构体来调用 CANopen 配置函数。一个表包含 4 个字节或少于 4 个字节的所有的只读、常量入口。第二个表包含所有变量和可写以及 SDO 分段的入口。

```
typedef struct _CAN_ODCONSTENTRY {
uint16_t index;
uint8_t subindex;
uint8_t len;
uint32_t val;
} CAN_ODCONSTENTRY;
//CAN_ODENTRY.entrytype_len 的高位半字节值
#define OD_NONE 0x00                //不存在对象字典入口
define OD_EXP_RO 0x10               //加速的对象字典入口，只读
#define OD_EXP_WO 0x20              //加速的对象字典入口，只写
#define OD_EXP_RW 0x30              //加速的对象字典入口，读写
#define OD_SEG_RO 0x40              //分段的对象字典入口，只读
#define OD_SEG_WO 0x50              //分段的对象字典入口，只写
#define OD_SEG_RW 0x60              //分段的对象字典入口，读-写
typedef struct _CAN_ODENTRY {
uint16_t index;
uint8_t subindex;
uint8_t entrytype_len;
uint8_t *val;
} CAN_ODENTRY;
typedef struct _CAN_CANOPENCFG {
uint8_t node_id;
```

```
uint8_t msgobj_rx;
uint8_t msgobj_tx;
uint32_t od_const_num;
CAN_ODCONSTENTRY *od_const_table;
uint32_t od_num;
CAN_ODENTRY *od_table;
} CAN_CANOPENCFG;
```

OD 表和 CANopen 配置结构体的范例：

```
//固定的、只读对象字典（OD）入口的列表
//只有加速的 SDO，长度为 1/2/4 字节
const CAN_ODCONSTENTRY myConstOD [] = {
//索引分索引长度值
{ 0x1000, 0x00, 4, 0x54534554UL }, //“测试”
{ 0x1018, 0x00, 1, 0x00000003UL },
{ 0x1018, 0x01, 4, 0x00000003UL },
{ 0x2000, 0x00, 1, (uint32_t)'M' },
};
//变量 OD 入口的列表
//加速的 SDO，长度为 1/2/4 字节
//分段的 SDO 应用程序处理，长度和值指针无关紧要
const CAN_ODENTRY myOD [] = {
//索引分索引访问类型|长度值指针
{ 0x1001, 0x00, OD_EXP_RO | 1, (uint8_t *)&error_register },
{ 0x1018, 0x02, OD_EXP_RO | 4, (uint8_t *)&device_id },
{ 0x1018, 0x03, OD_EXP_RO | 4, (uint8_t *)&fw_ver },
{ 0x2001, 0x00, OD_EXP_RW | 2, (uint8_t *)&param },
{ 0x2200, 0x00, OD_SEG_RW, (uint8_t *)NULL },
};
// CANopen 配置结构体
const CAN_CANOPENCFG myCANopen = {
20,                                          // node_id
5,                                           // msgobj_rx
6,                                           // msgobj_tx
sizeof(myConstOD)/sizeof(myConstOD[0]),      // od_const_num
(CAN_ODCONSTENTRY *)myConstOD,               // od_const_table
sizeof(myOD)/sizeof(myOD[0]),                // od_num
(CAN_ODENTRY *)myOD,                         // od_table
};
```

调用范例：

```
//初始化 CANopen 处理程序
(*rom)->pCAND->config_canopen((CAN_CANOPENCFG *)&myCANopen);
```

14.4.8 CANopen 处理程序

CANopen 处理程序处理 CANopen SDO 报文来访问对象字典，并在初始化时调用 CANopen 回调函数。只要应用程序有需要，就必须要经常周期性地调用该函数。

在 CANopen 应用程序中，SDO 处理通常具有最低的优先级，并在前台完成，而不是在中断进程中完成。

调用范例：

```
//调用 CANopen 处理程序
(*rom)->pCAND->canopen_handler();
```

14.4.9 CAN/CANopen 回调函数

CAN API 支持各种事件的回调函数。回调函数通过 API 函数来发布。

```
typedef struct _CAN_CALLBACKS {
void (*CAN_rx)(uint8_t msg_obj);
void (*CAN_tx)(uint8_t msg_obj);
void (*CAN_error)(uint32_t error_info);
uint32_t (*CANOPEN_sdo_read)(uint16_t index, uint8_t subindex);
uint32_t (*CANOPEN_sdo_write)(uint16_t index, uint8_t subindex, uint8_t *dat_ptr);
uint32_t (*CANOPEN_sdo_seg_read)(uint16_t index, uint8_t subindex, uint8_t openclose,
uint8_t *length, uint8_t *data, uint8_t *last);
uint32_t (*CANOPEN_sdo_seg_write)(uint16_t index, uint8_t subindex, uint8_t openclose,
uint8_t length, uint8_t *data, uint8_t *fast_resp);
uint8_t (*CANOPEN_sdo_req)(uint8_t length_req, uint8_t *req_ptr, uint8_t *length_resp,
 uint8_t *resp_ptr);
} CAN_CALLBACKS;
```

回调表定义的范例：

```
//回调函数指针的列表
const CAN_CALLBACKS callbacks = {
CAN_rx,
CAN_tx,
CAN_error,
CANOPEN_sdo_exp_read,
CANOPEN_sdo_exp_write,
CANOPEN_sdo_seg_read,
CANOPEN_sdo_seg_write,
CANOPEN_sdo_req,
};
```

调用范例：

```
//发布回调
(*rom)->pCAND->config_calb((CAN_CALLBACKS *)&callbacks);
```

14.4.10 CAN 报文接收的回调

CAN 中断处理程序会按照中断级别来调用 CAN 报文接收的回调函数。

调用范例：

```
//CAN 接收处理程序
void CAN_rx(uint8_t msgobj_num)
{
    //读出接收的报文
    msg_obj.msgobj = msgobj_num;
    (*rom)->pCAND->can_receive(&msg_obj);
    return;
}
```

注：如果用户 CANopen 处理程序是为用于 SDO 接收的报文对象激活时，不必调用回调函数。

14.4.11 CAN 报文发送回调

CAN 中断处理程序会按照中断级别来调用 CAN 报文发送回调函数。在产生 SDO 回应之前就要调用回调函数，以便允许修改或更新数据。

调用范例：

```
//CAN 发送处理程序
void CAN_tx(uint8_t msgobj_num)
{
  //复位应用程序使用的标志，等待发送完成
  if (wait_for_tx_finished == msgobj_num)
      wait_for_tx_finished = 0;
  return;
}
```

14.4.12 CAN 错误回调

CAN 中断处理程序会按照中断级别来调用 CAN 错误回调函数。

```
//错误状态位
#define CAN_ERROR_NONE 0x00000000UL
#define CAN_ERROR_PASS 0x00000001UL
#define CAN_ERROR_WARN 0x00000002UL
#define CAN_ERROR_BOFF 0x00000004UL
#define CAN_ERROR_STUF 0x00000008UL
#define CAN_ERROR_FORM 0x00000010UL
#define CAN_ERROR_ACK 0x00000020UL
#define CAN_ERROR_BIT1 0x00000040UL
#define CAN_ERROR_BIT0 0x00000080UL
#define CAN_ERROR_CRC 0x00000100UL
```

调用范例：

```
//CAN 错误处理程序
void CAN_error(uint32_t error_info)
//如果用户进入总线关闭状态，可以告诉应用程序重新初始化 CAN 控制器
if (error_info & CAN_ERROR_BOFF)
    reset_can = TRUE;
return;
```

14.4.13 CANopen SDO 加速的读回调

CANopen SDO 加速的读回调函数由 CANopen 处理程序调用。在产生 SDO 回应之前就要调用回调函数，以便允许修改或更新数据。

调用范例：

```
//用于加速读访问的 CANopen 回调
uint32_t CANOPEN_sdo_exp_read(uint16_t index, uint8_t subindex)
{
    //每次读取[2001h,0]，都会令 param 增加 1
    if ((index == 0x2001) && (subindex==0))
        param++;
    return 0;
}
```

14.4.14 CANopen SDO 加速的写回调

CANopen SDO 加速的写回调函数由 CANopen 处理程序调用。在写完新数据之前就要在新数据上传递回调函数，并调用回调函数，以允许反对或决定数据。

调用范例：

```
//用于加速写访问的 CANopen 回调
uint32_t CANOPEN_sdo_exp_write(uint16_t index, uint8_t subindex, uint8_t *dat_ptr)
{
  //将 0xAA55 写入到入口[2001h,0]以把写配置表的操作解锁
  if ((index == 0x2001) && (subindex == 0))
  if (*(uint16_t *)dat_ptr == 0xAA55)
  {
      write_config_ena = TRUE;
      return(TRUE);
   }
    else
      return(FALSE); //对其它值不予处理
}
```

14.4.15 CANopen SDO 分段的读回调

CANopen SDO 分段的读回调函数由 CANopen 处理程序调用。回调函数允许下列操作：

- 通知打开通读通道；
- 可向读取的主机提供高达七个字节的数据段；
- 当读取完所有数据时关闭通道；
- 可随时中止传送。

```
//用于 CANOPEN_sdo_seg_read/write() callback 'openclose'参数的值
#define CAN_SDOSEG_SEGMENT 0        //分段读/写
#define CAN_SDOSEG_OPEN 1           //通道打开
#define CAN_SDOSEG_CLOSE 2          //通道关闭
```

调用范例（读缓冲区）：

```
uint8_t read_buffer[0x123];
//用于分段读访问的 CANopen 回调
uint32_t CANOPEN_sdo_seg_read(uint16_t index, uint8_t subindex, uint8_t openclose,uint8_t *length, uint8_t *data, uint8_t *last)
{
    static uint16_t read_ofs;
    uint16_t i;
    if ((index == 0x2200) && (subindex==0))
    {
        if (openclose == CAN_SDOSEG_OPEN)
        {
            // Initialize the read buffer with "something"
            for (i=0; i<sizeof(read_buffer); i++)
            {
                read_buffer[i] = (i+5) + (i<<2);
            }
        read_ofs = 0;
        }
        else if (openclose == CAN_SDOSEG_SEGMENT)
        {
            i = 7;
            while (i && (read_ofs < sizeof(read_buffer)))
            {
                *data++ = read_buffer[read_ofs++];
                i--;
            }
            *length = 7-i;
            if (read_ofs == sizeof(read_buffer)) //读取整个缓冲区，这是最后的段区
            {
                *last = TRUE;
            }
        }
```

```
    return 0;
    }
    else
    {
      return SDO_ABORT_NOT_EXISTS;
    }
  }
```

14.4.16 CANopen SDO 分段的写回调

CANopen SDO 分段的写回调函数由 CANopen 处理程序调用。回调函数允许下列操作：

- 通知打开和关闭写通道；
- 可从写主机中传递高达七个字节的数据段；
- 当读取完所有数据时关闭通道；
- 可随时中止传送，例如，当存在缓冲区溢出时。

可以将响应选择为 8 字节（符合 CANopen 标准）或 1 字节（更快速，但是并不是所有的 SDO 客户端都支持）。

```
//用于 CANOPEN_sdo_seg_read/write() callback 'openclose'参数的值
#define CAN_SDOSEG_SEGMENT 0          //分段读/写
#define CAN_SDOSEG_OPEN 1             //通道打开
#define CAN_SDOSEG_CLOSE 2            //通道关闭
```

调用范例（写缓冲区）：

```
uint8_t write_buffer[0x321];
//用于分段写访问的 CANopen 回调
uint32_t CANOPEN_sdo_seg_write(uint16_t index, uint8_t subindex, uint8_t openclose,uint8_t length, uint8_t
*data, uint8_t *fast_resp)
{
    static uint16_t write_ofs;
    uint16_t i;
    if ((index == 0x2200) && (subindex==0))
    {
        if (openclose == CAN_SDOSEG_OPEN)
        {
            //初始化写缓冲区
            for (i=0; i<sizeof(write_buffer); i++)
            {
                write_buffer[i] = 0;
             }
             write_ofs = 0;
        }
        else if (openclose == CAN_SDOSEG_SEGMENT)
        {
            *fast_resp = TRUE;                          //使用快速 1 字节分段写响应
            i = length;
            while (i && (write_ofs < sizeof(write_buffer)))
```

```
                {
                    write_buffer[write_ofs++] = *data++;
                    i--;
                }
                if (i && (write_ofs >= sizeof(write_buffer)))          //太多数据要写
                {
                    return SDO_ABORT_TRANSFER;                         //不能被写入的数据
                }
            }
            else if (openclose == CAN_SDOSEG_CLOSE)
            {
                //已成功完成写入操作：标记缓冲区有效等等
            }
            return 0;
        }
    else
    {
        return SDO_ABORT_NOT_EXISTS;
    }
}
```

14.4.17 CANopen fall-back SDO 处理程序回调

CANopen fall-back SDO 处理程序回调函数由 CANopen 处理程序调用。只要存在着一个不能被处理的 SDO 请求或 SDO 请求被 SDO 中止响应结束，都要调用该函数。调用时要使用请求的完整数据缓冲区，并允许产生任何类型的 SDO 响应。这可用于执行定制的 SDO 处理程序，例如，执行 SDO 模块传输方式。

```
//返回用于 CANOPEN_sdo_req() callback 的值
#define CAN_SDOREQ_NOTHANDLED 0          //正常处理，无影响
#define CAN_SDOREQ_HANDLED_SEND 1        //回调处理，自动发送
#define CAN_SDOREQ_HANDLED_NOSEND 2      //回调处理，不发送
```