[对象字典 2](#_Toc476924861)

[创建对象字典 2](#_Toc476924862)

[查找对象 5](#_Toc476924863)

[访问对象字典 7](#_Toc476924864)

[CAN控制器 9](#_Toc476924865)

[时间管理 10](#_Toc476924866)

[定时器 10](#_Toc476924867)

[创建定时任务 11](#_Toc476924868)

[定时过程 13](#_Toc476924869)

[NMT 14](#_Toc476924870)

[设备状态 14](#_Toc476924871)

[状态转换 15](#_Toc476924872)

[配置 16](#_Toc476924873)

[发送NMT报文 17](#_Toc476924874)

[接收NMT报文 17](#_Toc476924875)

[NEC 18](#_Toc476924876)

[配置心跳报文 18](#_Toc476924877)

[配置节点保护 19](#_Toc476924878)

[发送NEC报文 20](#_Toc476924879)

[接收NEC报文 21](#_Toc476924880)

[启动传输 22](#_Toc476924881)

[EMCY 23](#_Toc476924882)

[应急错误 23](#_Toc476924883)

[发送EMCY报文 24](#_Toc476924884)

[接收EMCY报文 25](#_Toc476924885)

[启动传输 26](#_Toc476924886)

[SDO 27](#_Toc476924887)

[创建SDO服务器 27](#_Toc476924888)

[配置SDO服务器 29](#_Toc476924889)

[查看SDO服务器 30](#_Toc476924890)

[发送SDO报文 31](#_Toc476924891)

[SDO请求 33](#_Toc476924892)

[SDO传输 34](#_Toc476924893)

[SYNC 35](#_Toc476924894)

[配置 35](#_Toc476924895)

[发送SYNC报文 36](#_Toc476924896)

[启动传输 37](#_Toc476924897)

[PDO 38](#_Toc476924898)

[PDO服务器 38](#_Toc476924899)

[发送PDO报文 39](#_Toc476924900)

[接收PDO报文 42](#_Toc476924901)

[同步触发 43](#_Toc476924902)

[远程帧触发 44](#_Toc476924903)

[启动传输 45](#_Toc476924904)

[LSS 46](#_Toc476924905)

[参数配置 46](#_Toc476924906)

# 对象字典

## 创建对象字典

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 一个条目 | subindex | 成员 | 类型 | 说明 | | | |
| bAccessType | UNS8变量 | 该条目的访问类型 | | RW | 读/写 |
| WO | 只写 |
| RO | 只读 |
| bDataType | UNS8变量 | 该条目的数据类型 | | boolean | 布尔型 |
| int8 | 8位无符号整型 |
| …… | …… |
| identity | id |
| size | UNS32变量 | 存储该条目内容的变量 的 数据类型 的 位宽  如：存储该条目内容的变量的数据类型是UNS8，那么  该size成员的值就是sizeof (UNS8) | | | |
| pObject | void指针 | 存储该条目内容的变量的地址 | | | |
|  | | | | | | | |
| 一个对象 | indextable | pSubindex | subindex指针 | 该对象的第一个条目的地址，即该对象的条目数组的首地址；  每一个对象对应一个条目数组，即subindex类型的数组，数组元素就是该对象的条目；  条目的子索引号就是该条目在条目数组中的序号 | | | |
| bSubCount | UNS8变量 | 该对象包含的 子索引（条目）的总数（包括第一个条目），  范围：[1,255] | | | |
| index | UNS16变量 | 该对象的16位索引号 | | | |
|  | | | | | | | |
| 快速  对象组 | quick\_index | SDO\_SVR | UNS16 | SDO server parameter对象 在对象字典数组中的序号 | | | |
| SDO\_CLT | UNS16 | SDO client parameter对象 在对象字典数组中的序号 | | | |
| PDO\_RCV | UNS16 | RPDO communication parameter对象 在对象字典数组中的序号 | | | |
| PDO\_RCV\_MAP | UNS16 | RPDO mapping parameter对象 在对象字典数组中的序号 | | | |
| PDO\_TRS | UNS16 | TPDO communication parameter对象 在对象字典数组中的序号 | | | |
| PDO\_TRS\_MAP | UNS16 | TPDO mapping parameter对象 在对象字典数组中的序号 | | | |
|  | | | | | | | |
| 对象  回调函数  类型 | ODCallback\_t | 输入 | CO\_Data指针 | | ODCallback\_t用于定义一个指针变量  变量赋值某个函数的地址(即函数名)  要求该函数的输入输出满足ODCallback\_t； | | |
| indextable指针 | |
| UNS8变量 | |
| 输出 | UNS32变量 | |

|  |  |
| --- | --- |
| 创建  一个对象 | UNS8 ObjDict\_highestSubIndex\_obj60C1 = 3; /\*存储第一个条目内容的变量，即本对象子索引（条目）总数\*/  UNS8 InterpolationDataIndexNum = 2; /\*存储第二个条目内容的变量 \*/  INTEGER32 InterpolationDataAxis1 = 0; /\*存储第三个条目内容的变量 \*/  INTEGER32 InterpolationDataAxis2 = 0; /\*存储第四个条目内容的变量 \*/  subindex ObjDict\_Index60c1[] = /\*subindex类型数组，表示一个对象的本体\*/  {  { RW, int8, sizeof (UNS8), (void\*)&ObjDict\_highestSubIndex\_obj60C1 }, //第一个条目  { RW, int8, sizeof (UNS8), (void\*)&InterpolationDataIndexNum }, //第二个条目  { RW, int32, sizeof (UNS32), (void\*)&InterpolationDataAxis1 }, //第三个条目  { RW, int32, sizeof (UNS32), (void\*)&InterpolationDataAxis2 }, //第四个条目  }; |
| 创建  回调函数  数组 | 有些对象在创建时附带创建一个回调函数数组；**即一个对象对应一个ODCallback\_t数组，称为回调函数数组**；  数组的每一个元素对应一个条目；每一个元素是其对应条目的回调函数的地址  如果某个条目没有回调函数，那么该条目在回调函数数组中对应的元素的值是NULL |
| 创建  对象字典 | const indextable ObjDict\_objdict[] =  {  { (subindex\*)ObjDict\_Index1000,sizeof(ObjDict\_Index1000)/sizeof(ObjDict\_Index1000[0]), 0x1000},  { (subindex\*)ObjDict\_Index1001,sizeof(ObjDict\_Index1001)/sizeof(ObjDict\_Index1001[0]), 0x1001},  { (subindex\*)ObjDict\_Index1005,sizeof(ObjDict\_Index1005)/sizeof(ObjDict\_Index1005[0]), 0x1005},  …………  创建indextable类型数组，表示该设备的对象字典；又称为对象字典数组  只要知道某个对象在对象字典中的序号，就可以获得这个对象，如本例中索引号为1005的对象就是： ObjDict\_objdict[3] |
| 创建  快速对象组 | 快速对象组是对用户自己创建的**对象字典数组**的一种应用方法  快速对象组用来方便用户快速得到**SDO参数对象** **和 PDO参数对象**；  快速对象组仅仅是为了提供便利，其功能是多余的，因为用户可以直接通过对象字典数组来找到相应的对象；  要注意的是：快速对象组数据类型quick\_index，其结构成员表示的是相应的对象在对象字典数组中的序号，而不是对象的索引；创建快速对象组的时候一定要仔细对照对象字典数组来配置，两者要相一致。  要求至少创建两个快速对象组，分别传递给CANopen设备的firstIndex成员和lastIndex成员 |
| 创建快速对象组ObjDict\_firstIndex，包含：**对象字典中排在最前面的**SDO服务端参数对象、SDO客户端参数对象、  TPDO通讯参数对象、TPDO映射参数对象、RPDO通讯参数对象、RPDO映射参数对象，那么：  const quick\_index ObjDict\_firstIndex = {  8, /\*SDO1 服务端参数🡪ObjDict\_objdict[ObjDict\_firstIndex.SDO\_SVR]  9, /\*SDO1 客户端参数🡪ObjDict\_objdict[ObjDict\_firstIndex.SDO\_CLT]  10, /\* RPDO1 通讯参数对象🡪ObjDict\_objdict[ObjDict\_firstIndex.PDO\_RCV]  12, /\* RPDO1映射参数对象🡪ObjDict\_objdict[ObjDict\_firstIndex.PDO\_RCV\_MAP]  14, /\* TPDO1通讯参数对象🡪ObjDict\_objdict[ObjDict\_firstIndex. PDO\_TRS]  16 /\* TPDO1 映射参数对象🡪ObjDict\_objdict[ObjDict\_firstIndex. PDO\_TRS\_MAP]  }; |
| 创建快速对象组ObjDict\_lastIndex，包含：**对象字典中排在最后的**SDO服务端参数对象、SDO客户端参数对象、  TPDO通讯参数对象、TPDO映射参数对象、RPDO通讯参数对象、RPDO映射参数对象：  const quick\_index ObjDict\_lastIndex = {  8, /\*SDO1服务端参数对象🡪ObjDict\_objdict[ObjDict\_firstIndex.SDO\_SVR]  9, /\*SDO1客户端端参数对象🡪ObjDict\_objdict[ObjDict\_firstIndex.SDO\_CLT]  11, /\* RPDO2通讯参数对象🡪ObjDict\_objdict[ObjDict\_firstIndex.PDO\_RCV]  13, /\* RPDO2映射参数对象🡪ObjDict\_objdict[ObjDict\_firstIndex.PDO\_RCV\_MAP]  15, /\* TPDO2通讯参数对象🡪ObjDict\_objdict[ObjDict\_firstIndex. PDO\_TRS]SDO1  17 /\* TPDO2映射参数对象🡪ObjDict\_objdict[ObjDict\_firstIndex. PDO\_TRS\_MAP]  };  🡪两个快速对象组，可以包含重复的对象，由于该CANopen设备仅配置了一个SDO服务器，即SDO1，  所以SDO1服务端参数对象和SDO1客户端端参数对象 既是排在第一的，也是排在倒数第一的；  🡪如果知道两个对象在对象字典中的间隔，那么在已得到其中一个对象的情况下，另一个对象也能马上得到，  如已知TPDO1通讯参数对象：ObjDict\_objdict[ObjDict\_firstIndex. PDO\_TRS]  用户知道TPDO1通讯参数对象和TPDO2通讯参数对象 两个对象是相邻的，那么直接就可以得到：  TPDO2通讯参数对象：ObjDict\_objdict[ObjDict\_firstIndex. PDO\_TRS+1] |
| 对象字典  的容量 | 创建一个UNS16类型的变量  表示本设备的对象字典包含的对象总数 |

|  |  |  |  |
| --- | --- | --- | --- |
| CO\_Data | 成员 | 类型 | 配置 |
| objdict | indextable指针 | 创建indextable数组，即对象字典，传递数组首地址 |
| firstIndex | quick\_index指针 | 创建一个快速对象组，数组元素包含：**对象字典中排在最前面的**  SDO服务端参数对象、SDO客户端参数对象、TPDO通讯参数对象、  TPDO映射参数对象、RPDO通讯参数对象、RPDO映射参数对象；  传递该数组的首地址 |
| lastIndex | quick\_index指针 | 创建一个快速对象组，数组元素包含：**对象字典中排在最后面的**  SDO服务端参数对象、SDO客户端参数对象、TPDO通讯参数对象、  TPDO映射参数对象、RPDO通讯参数对象、RPDO映射参数对象；  传递该数组的首地址 |
| ObjdictSize | UNS16指针 | 创建UNS16变量存储对象字典的容量，传递变量地址 |
| valueRangeTest | valueRangeTest\_t  变量 | 创建scanIndexOD\_t类型的函数，传递该函数的地址，即函数名 |
| scanIndexOD | scanIndexOD\_t  变量 | 创建scanIndexOD\_t类型的函数，传递该函数的地址，即函数名 |
| storeODSubIndex | storeODSubIndex\_t  变量 | 创建scanIndexOD\_t类型的函数，传递该函数的地址，即函数名 |

## 查找对象

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| scanIndexOD | 输入 | UNS16变量 | 输入对象的索引号 | | |
| UNS32指针 | 创建一个空白的UNS32类型的变量，将该变量的地址传递给本函数  函数执行过程中会将错误信息存储在该变量中 | | |
| OD\_Callback\_t  指针的指针 | 如果输入的对象附带有回调函数，那么在该参数输入  该对象的 **回调函数数组** 的**首地址**； | | |
| 输出 | indextable  指针 | 根据输入的索引号，输出其对应的对象，即indextable变量的地址；  输出NULL：表明查遍整个对象字典，没有找到对应的对象 | | |
| 错误信息 | OD\_NO\_SUCH\_OBJECT | | 查遍整个对象字典，没有找到对应的对象 |
| OD\_SUCCESSFUL | | 查找成功 |
| 使用 | 本函数用于获取 指定索引 的 对象； | | | |
| storeODSubIndex | 输入 | CO\_Data指针 |  | | |
| UNS16变量 |  | | |
| UNS8变量 |  | | |
| valueRangeTest | 输入 | UNS8变量 |  | | |
| void指针 |  | | |
| 输出 | 0 | |  | |
| OD\_VALUE\_RANGE\_EXCEEDED | |  | |
| 使用 | 该函数用于测试 | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 创建  scanIndexOD\_t  类型的函数 | 输入 | UNS16变量 | 输入对象的索引号 | | |
| UNS32指针 | 预先创建一个UNS32类型的变量，将该变量的地址传递给本函数  函数执行过程中会将错误信息存储在该变量中 | | |
| OD\_Callback\_t  指针的指针 | 如果输入的对象附带有回调函数，那么在该参数输入  该对象的 **回调函数数组** 的**首地址**； | | |
| 输出 | indextable  指针 | 根据输入的索引号，输出其对应的对象，即indextable变量的地址；  输出NULL：表明查遍整个对象字典，没有找到对应的对象 | | |
| 错误信息 | OD\_NO\_SUCH\_OBJECT | | 查遍整个对象字典，没有找到对应的对象 |
| OD\_SUCCESSFUL | | 查找成功 |
| 使用 | 本函数用于获取 指定索引 的 对象； | | | |
| 创建  storeODSubIndex\_t  类型的函数 | 输入 | CO\_Data指针 |  | | |
| UNS16变量 |  | | |
| UNS8变量 |  | | |
| 创建  valueRangeTest\_t  类型的函数 | 输入 | UNS8变量 |  | | |
| void指针 |  | | |
| 输出 | 0 | |  | |
| OD\_VALUE\_RANGE\_EXCEEDED | |  | |
| 使用 | 该函数用于测试 | | | |

## 访问对象字典

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 读取  对象  字典 | \_getODentry | 输入 | CO\_Data指针 | Canopen设备地址 | | | | |
| UNS16变量 | 指定读取的对象的索引号 | | | | |
| UNS8变量 | 指定读取的条目的子索引号 | | | | |
| void指针 | 预先创建一个变量，输入该变量的地址  函数执行过程中将制定读取的条目的内容存储在该变量中； | | | | |
| UNS32指针 |  | | | | |
| UNS8指针 | 0：  1： | | | | |
| UNS8变量 | WO | | |  | |
| RO | | |  | |
| RW | | |  | |
| UNS8变量 | 0：little endian  1：big endian | | | | |
| 输出 | OD\_SUCCESSFUL | | | |  | |
| OD\_READ\_NOT\_ALLOWED | | | |  | |
| OD\_WRITE\_NOT\_ALLOWED | | | |  | |
| OD\_NO\_SUCH\_OBJECT | | | |  | |
| OD\_NOT\_MAPPABLE | | | |  | |
| OD\_LENGTH\_DATA\_INVALID | | | |  | |
| OD\_NO\_SUCH\_SUBINDEX | | | |  | |
| OD\_VALUE\_RANGE\_EXCEEDED | | | |  | |
| OD\_VALUE\_TOO\_LOW | | | |  | |
| OD\_VALUE\_TOO\_HIGH | | | |  | |
| SDOABT\_GENERAL\_ERROR | | | |  | |
| SDOABT\_OUT\_OF\_MEMORY | | | |  | |
| 使用 |  | | | | | |
| getODentry | #define getODentry( OD, wIndex, bSubindex, pDestData, pExpectedSize, pDataType, checkAccess)  \_getODentry( OD, wIndex, bSubindex, pDestData, pExpectedSize,pDataType, checkAccess, **1**) | | | | | | |
| readLocalDict | #define readLocalDict( OD, wIndex, bSubindex, pDestData, pExpectedSize, pDataType, checkAccess)  \_getODentry( OD, wIndex, bSubindex, pDestData, pExpectedSize, pDataType, checkAccess, **0**) | | | | | | |
|  | | | | | | | | |
| 写入  对象  字典 | \_setODentry | 输入 | CO\_Data指针 | | Canopen设备地址 | | | |
| UNS16变量 | | 指定写入的对象的16位索引 | | | |
| UNS8变量 | | 指定写入的条目的8位子索引 | | | |
| void指针 | | 预先创建一个变量，输入该变量的地址，  该变量存储要编辑的内容 | | | |
| UNS32指针 | | 预先创建一个变量，输入该变量的地址，  该变量存储辑内容的字节数 | | | |
| UNS8变量 | | RW | 设置该条目的访问类型：读/写 | | |
| WO | 设置该条目的访问类型：只写 | | |
| RO | 设置该条目的访问类型：只读 | | |
| UNS8变量 | | 0：  1： | | | |
| 输出 | UNS32变量 | | OD\_SU CCESSFUL | | |  |
| SDOABT\_TOGGLE\_NOT\_ALTERNED | | |  |
| SDOABT\_TIMED\_OUT | | |  |
| SDOABT\_OUT\_OF\_MEMORY | | |  |
| SDOABT\_GENERAL\_ERROR | | |  |
| SDOABT\_LOCAL\_CTRL\_ERROR | | |  |
| 使用 |  | | | | | |
| setODentry | #define setODentry( d, wIndex, bSubindex, pSourceData, pExpectedSize, checkAccess)  \_setODentry( d, wIndex, bSubindex, pSourceData, pExpectedSize, checkAccess, **1**) | | | | | | |
| writeLocalDict | #define writeLocalDict( d, wIndex, bSubindex, pSourceData, pExpectedSize, checkAccess)  \_setODentry( d, wIndex, bSubindex, pSourceData, pExpectedSize, checkAccess, **0**) | | | | | | |

配置回调函数

|  |  |  |  |
| --- | --- | --- | --- |
| RegisterSetODentryCallBack | 输入 | CO\_Data指针 | Canopen设备地址 |
| UNS16变量 |  |
| UNS8变量 |  |
| ODCallback\_t变量 |  |
| 输出 | UNS32变量 |  |
| 使用 |  | |

# CAN控制器

|  |  |  |  |
| --- | --- | --- | --- |
|  | 成员 | 类型 |  |
| Message | cob\_id | UNS16 | 只使用低11位，用于存储这个报文的COB-ID |
| rtr | UNS8 | 1：RTR =1 →该报文是远程帧  0：RTR=0 →该报文是数据帧 |
| len | UNS8 | 该报文的数据场包含的字节个数；即can报文DLC段的值 |
| data[8] | UNS8 | CAN报文的数据场内容 |
|  | | | |
| canSend | 输入 | CAN\_PORT  变量 | CAN1 或 CAN2 |
| Message  指针 | 创建一个Message变量，表示一个CAN报文；  输入该Message变量的地址； |
| 定义 | 使用canfestival程序所在的芯片自带的can控制器发送CAN报文； | |

使用CAN中断，在CAN中断服务程序中调用canDispatch函数

|  |  |  |  |
| --- | --- | --- | --- |
| canDispatch | 输入 | CO\_Data  指针 | Canopen设备地址 |
| Message  指针 | 发送的消息 |
| 输出 |  | |
| 使用 | 设备一旦收到CAN报文，就会进入CAN中断，然后执行CAN接收中断服务函数中的canDispatch()函数  函数会查看所接收到的CAN报文的CAN-ID的4位功能码，获知接收报文是哪种canopen报文  针对不同的报文，有对应的处理函数： | |
| NMT报文 | 调用proceedNMTstateChange()函数 |
| SYNC报文 | 调用SYNCproceed()函数 |
| PDO报文 | 调用PDOproceed()函数 |
| SDO报文 | 调用SDOproceed()函数 |
| NEC报文 | 调用proceedNODE\_GUARD()函数 |
| LSS报文 | 调用SYNCproceed()函数 |

|  |  |  |  |
| --- | --- | --- | --- |
| CO\_Data | 成员 | 类型 | 配置 |
| canHandle | CAN\_PORT变量 |  |

# 时间管理

## 定时器

|  |  |  |  |
| --- | --- | --- | --- |
| 配置定时器 | 整个canfestival程序只需要使用到一个定时器，即程序所使用的芯片提供的定时器外设；  由这个定时器来驱动所有的定时器任务；  该定时器在canfestival程序初始化时配置完毕，并直接进入工作状态；  在canfestival程序运行的过程中，该定时器始终处于连续的工作状态； | | |
| 计数频率 | 用户要知道所使用的定时器的计数器的计数频率；  定时计数器节拍 =1/计数频率 = 计数周期 | |
| setTimer | 输入 | TIMEVAL  变量 | 即输入一个32位无符号整型变量；  设置定时器的计时时间长度；  单位：定时计数器节拍 |
| 使用 | setTimer()函数用于启动定时任务，主要是在两个地方被调用  (1)在SetAlarm()函数中调用setTimer()函数  (2)在TimeDispatch()函数中调用setTimer()函数  从调用本函数的时刻开始，经过输入的 定时器计时时间长度，定时器溢出；  如：在定时计数器值为500时，调用本函数setTimer(0x3E8); 那么：  定时计数器的数值达到1500时(假设最大计数值大于1500)，该任务溢出，  然后计数器返回到0(假设定时器是从0开始向上计数)，重新开始计数  函数的调用不影响定时器的工作状态； | |
| 中断服务程序 | 定时任务  溢出时刻 | 定时任务溢出时，执行中断服务函数； | |
| 定时任务  TimeDispatch  时刻 | 每次发生定时器溢出时，就会立刻返回到初始值重新开始计数，**然后**在中断服务程序中调用TimeDispatch()函数；中断服务程序中可能包含其他内容，所以定时器溢出的时刻和相应地执行TimeDispatch()函数之间会有时间差；  我们紧邻着TimeDispatch()函数获取当前的定时计数器值last\_time\_set，记录执行TimeDispatch的时刻，所以加下来我们所讲到的TimeDispatch时刻，都指的是TimeDispatch()函数刚开始执行的时刻；近似就认为是定时任务执行的时刻 | |
| getElapsedTime | 输出 | TIMEVAL  变量 | 函数返回一个32位无符号整型变量；  **此刻** 距离 **上一次TimeDispatch** 的时间长度，单位：定时计数节拍  返回 return timer > last\_time\_set ? timer - last\_time\_set : last\_time\_set - timer;  如果是向上计数：返回的是timer - last\_time\_set  如果是向下计数：返回的是last\_time\_set – timer |

## 创建定时任务

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| s\_timer\_entry | 成员 | 类型 | 说明 | | |
| state | UNS8变量 | TIMER\_FREE | | 空闲状态 |
| TIMER\_ARMED | |  |
| TIMER\_TRIG | |  |
| TIMER\_TRIG\_PERIOD | |  |
| d | CO\_Data指针 | 该定时任务所属的设备的地址 | | |
| callback | TimerCallback\_t  变量 | 每一个定时任务都可以配置一个回调函数  每当该定时任务溢出，都会以执行它的回调函数 | | |
| id | UNS32变量 | 本定时任务的id； | | |
| val | TIMEVAL变量 | **上次定时器溢出执行TimeDispatch的时刻，到该定时任务溢出时刻，的时间长度**  单位：定时器计数节拍 | | |
| 初始化 | 调用SetAlarm()函数创建一个定时任务，初始化这个定时任务的val  一个定时任务的val只会初始化一次 | |
| 更新 | 在TimeDispatch()函数中，所有处于ARMED状态的定时任务都会更新各自的val成员的值 | |
| interval | TIMEVAL变量 | interval ≠ 0，表明该定时任务是周期任务；interval就是它的溢出周期；  interval是该定时任务相邻两次TimeDispatch的时间间隔  从该任务第一次TimeDispatch开始，每经过interval个定时计数节拍该任务就会再次执行TimeDispatch()函数；  第一次溢出的时刻根据其(创建该定时任务时所配置的)val初始值确定；  以后该任务溢出的时刻(以上一次定时器溢出为起点)需要根据interval成员来计算，然后将计算结果存储到其val成员中；  interval = 0，表明该定时任务不是周期任务，它只执行一次； | | |

|  |  |  |
| --- | --- | --- |
| timers[MAX\_NB\_TIMER] | s\_timer\_entry  变量 | 创建定时任务组  系统就创建定时任务组timers[MAX\_NB\_TIMER]，即创建MAX\_NB\_TIMER个定时任务  MAX\_NB\_TIMER定义位置：config.h  定时任务的序号就是该定时任务在定时任务组中对应的索引；  这些定时任务拥有相同的初始值：{TIMER\_FREE, NULL, NULL, 0, 0, 0}，  这些任务在最开始的时候就创建了，但都处于空闲状态；等待被启动 |
| TIMEVAL\_MAX | define  宏定义 | 定时计数器的计数上限；  定义位置：timercfg.h |
| total\_sleep\_time | TIMEVAL  变量 | total\_sleep\_time是一个全局变量，表示所有定时器任务中，最早溢出的那个任务的val  更新： |
| 执行SetAlarm()函数创建新的定时任务，有新的val加入，  比较此刻的total\_sleep\_time与新加入的val的大小  total\_sleep\_time = min(total\_sleep\_time , val)，表示的是 |
| TimeDispatch()函数中所有定时任务的val都会更新  每个val更新完毕后都会与此刻的total\_sleep\_time比较大小  total\_sleep\_time = min(total\_sleep\_time , sleep\_time) |
| last\_timer\_raw | TIMER\_HANDLE  变量 | 定时任务组中排在最后面的非空闲定时任务的序号；  初始情况下，系统中没有定时任务，所以设置last\_timer\_raw的值为-1；  如：last\_timer\_raw=6，那么表示timers[6]肯定时处于工作状态，  而序号在 (6,MAX\_NB\_TIMER) 的所有定时任务都处于空闲状态state=TIMER\_FREE |

|  |  |  |  |
| --- | --- | --- | --- |
| SetAlarm | 输入 | CO\_Data指针 | Canopen设备地址 |
| UNS32 | 输入该定时任务 的序号 |
| TimerCallback\_t | 输入该定时任务 的回调函数的地址(函数名) |
| TIMEVAL | 输入该定时任务 定时的时间长度timer\_value  即创建该定时任务的时刻 到 该任务定时溢出 的时间间隔； |
| TIMEVAL | 输入该定时任务 的周期 |
| 定义 | 本函数用于创建一个定时任务，即配置一个空闲的s\_timer\_entry变量，并使其进入工作状态；  那么，**调用本函数的时刻就是该定时任务开始计时的时刻** | |
| 寻找空闲任务 | 函数从timers[0]开始，遍历整个定时任务组，  直到遇到一个state== TIMER\_FREE的定时任务；  该任务将接受本函数的配置，进入工作状态； |
| val  初始化 | 上次TimeDispatch 到 该任务的定时创建启动 的时间间隔：getElapsedTime()  该定时任务创建时刻 到 该定时任务溢出的时间间隔：timer\_value  **上次定时器溢出的TimeDispatch 到 该定时任务溢出 的时间间隔**：  val= getElapsedTime() + timer\_value ，单位：定时计数器节拍 |
| 更新  total\_sleep\_time | 前面讲到total\_sleep\_time总是所有定时任务中，最早执行完毕的那个任务的sleep\_time  比较 total\_sleep\_time 与val  🡪如果**val＜total\_sleep\_time**，那么：  该新定时器任务执行完毕的时刻到来的更快，所以需要更新total\_sleep\_time， |
| 🡪如果**val≥total\_sleep\_time**，那么：  已经创建的定时器任务中，有比新定时器任务的任务执行完毕时刻更早的任务，  total\_sleep\_time和定时器配置都保持不变 |
| 启动计时 | 调用SetTimer函数，输入该定时任务的val，更新定时器； |
| 配置定时器任务 | 配置该定时任务(s\_timer\_entry变量)的其他各个成员---根据本函数的输入参数；  该定时任务的状态state变为：TIMER\_ARMED ，进入工作状态  val成员配置为getElapsedTime() + timer\_value |
| DelAlarm | 输入 | TIMER\_HANDLE  变量 | 输入要终止的目标定时任务的序号 |
| 定义 | 本函数用于终止一个定时任务，将该定时任务的状态变为空闲状态TIMER\_FREE | |

## 定时过程

|  |  |  |  |
| --- | --- | --- | --- |
| TimeDispatch | 前面讲到，**定时时间是从该定时任务开始计时，到执行该定时任务；**  **一般情况下就认为是从该定时任务开始计时，到执行该定时任务的TimeDispatch；**  但TimeDispatch()函数的要求更为严格，因为TimeDispatch()函数也有许多其他的代码段，真正执行该定时任务是其中之一，在TimeDispatch()函数中调用getElapsedTime()：  **overrun = getElapsedTime()：**本次定时任务溢出到 本次TimeDispatch 的 时间长度  **real\_total\_sleep\_time = total\_sleep\_time+ overrun** ：上次TimeDispatch时刻 到本次TimeDispatch的时间长度  **total\_sleep\_time：**刚开始表示上次TimeDispatch时刻 到 本次定时器溢出的时间长度  一旦被更新，它就表示本次TimeDispatch到下次TimeDispatch 的时间长度 | | |
| 定时任务溢出 | 遍历整个定时任务组，查看每个任务的val成员：  如果**val ≤ real\_total\_sleep\_time** ，这里其实又分了两种情况：  第一种情况：**val = total\_sleep\_time，**该定时器任务就是对应本次发生定时器溢出  第二种情况：**total\_sleep\_time＜val ≤real\_total\_sleep\_time，**本次定时器溢出时是因为另外某一个更早的定时任务(称为任务X)；**而本定时任务的溢出时刻是在定时任务X的溢出时刻到定时任务X的TimeDispatch时刻之间**  上述两种情况都表明该定时任务已经溢出；后续操作是相同的；  对于发生溢出的定时任务，要查看它的interval成员，interval是该定时任务相邻两次执行TimeDispatch的间隔 | |
| 单次  任务 | interval=0：表明该定时任务不是周期任务，  它只溢出一次(就是本次)，配置该任务进入Trig状态； |
| 周期  任务 | interval≠0：表明该定时任务是周期任务；  更新该定时任务的val成员(以本次TimeDispatch时刻为起点)：  val = interval - (overrun % interval)  然后更新total\_sleep\_time = min(total\_sleep\_time , val)  配置该任务进入Trig\_Period状态； |
| 定时任务未溢出 | 如果**val＞real\_total\_sleep\_time**，表明：  该定时任务还没到达其溢出的时刻，  更新该定时任务的val成员(以本次TimeDispatch为起点)：val = val – real\_total\_sleep\_time  然后更新total\_sleep\_time = min(total\_sleep\_time , val) | |
| 调用SetTimer()，输入最新的total\_sleep\_time | | |
| 再次遍历整个定时任务组，查看每个任务的状态 | | |

# NMT

## 设备状态

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| CO\_Data: | | | | | |
| 成员 | | 类型 | 说明 | | | |
| bDeviceNodeId | | UNS8指针 | 创建UNS8变量存储该设备NodeID，传递变量地址 | | | |
| iam\_a\_slave | | UNS8指针 | 创建UNS8变量存储该设备的主/从属性，传递变量地址  1🡪从机 ； 0🡪主机 | | | |
| nodeState  设备当前节点状态 | | e\_nodeState  变量 | Initialisation | | 初始化状态 | |
| Disconnected | | 未连接状态 | |
| Connecting | | 连接状态 | |
| Preparing | | 准备状态 | |
| Stopped | | 停止状态 | |
| Operational | | 运行状态 | |
| Pre\_operational | | 预操作状态 | |
| Unknown\_state | | 未知状态 | |
| CurrentCommunicationState  设备当前通讯状态 | | s\_state\_communication  变量 | 如：\*->s\_state\_communication ={0,0,1,0,0,0,0} ，表示设备当前的通讯状态是：只能进行EMCY传输，其他传输都禁止  创建CANopen设备时的初始通讯状态是{0,0,0,0,0,00} | | | |
| 成员 | 类型 | | 说明 |
| csBoot\_Up | INTEGER8  变量 | | 1：允许boot报文传输  0：禁止boot报文传输 |
| csSDO | 1：允许sdo报文传输  0：禁止sdo报文传输 |
| csEmergency | …… |
| csSYNC | …… |
| csHeartbeat | …… |
| csPDO | …… |
| csLSS | …… |
| initialisation | | initialisation\_t变量 | typedef void (\*initialisation\_t)(CO\_Data\*);  创建上述类型函数，传递函数地址，即函数名 | | | |
| preOperational | | preOperational\_t变量 | typedef void (\*preOperational\_t)(CO\_Data\*);  创建上述类型函数，传递函数地址，即函数名 | | | |
| operational | | operational\_t变量 | typedef void (\*operational\_t)(CO\_Data\*);  创建上述类型函数，传递函数地址，即函数名 | | | |
| stopped | | stopped\_t变量 | typedef void (\*stopped\_t)(CO\_Data\*);  创建上述类型函数，传递函数地址，即函数名 | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| getNodeId | 输入 | CO\_Data指针 | Canopen设备地址 |
| 输出 | return \*->bDeviceNodeId  输出设备的**节点ID** | |
| setNodeId | 输入 | CO\_Data指针 | Canopen设备地址 |
| UNS8变量 | 为本设备设置的目标节点ID  范围：[1,127] |
| getState | 输入 | CO\_Data指针 | Canopen设备地址 |
| 输出 | return \*->nodeState  输出设备的**节点状态** | |

## 状态转换

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| switchCommunicationState | | | 输入 | CO\_Data指针 | | | Canopen设备地址 | | | |
| s\_state\_communication  变量 | | | 输入要为本设备设置的目标通讯状态newCommunicationState  举例说明： | | | |
| {1,0,0,0,0,0} | | 启动boot报文传输，  关闭其他通讯 | |
| {0,1,1,1,1,0} | | 启动SDO、EMCY、SYNC、HeartBeat  禁止Boot\_Up、LSS | |
| …… | | …… | |
| 使用 | 调用宏函数：StartOrStop()，**查看每一个通讯对象：**  如测试SDO，即StartOrStop(csSDO,None,resetSDO(d)) ：  查看该通讯对象的当前通讯状态：\*d->CurrentCommunicationState.csSDO  获取要设置的目标通讯状态：newCommunicationState.csSDO  当前通讯状态=0，目标通讯状态=1：执行该通信对象的启动函数  当前通讯状态=1，目标通讯状态=0：执行该通信对象的停止函数 | | | | | | |
| 通讯 | | | | 启动传输 | | 停止传输 |
| boot\_up | | | | / | | SlaveSendBootUp() |
| sdo | | | | / | | resetSDO() |
| emcy | | | | emergencyInit() | | emergencyStop() |
| sync | | | | startSYNC() | | stopSYNC() |
| heartbeat | | | | heartbeatInit() | | heartbeatStop() |
| pdo | | | | PDOInit() | | PDOStop() |
| lss | | | | startLSS() | | stopLSS() |
|  | | | | | | | | | | |
| setState | 输入 | CO\_Data指针 | | | Canopen设备地址 | | | | | |
| e\_nodeState  变量 | | | Initialisation | 函数使用switch语句  根据输入的目标节点状态，执行相应的操作 | | | | |
| Pre-operational |
| Operational |
| Stopped |
| 输出 | 设置完成后输出设备的节点状态，用于检验。 | | | | | | | | |
| 使用 | Initialisation | | | \*->nodeState=Initialisation  调用switchCommunicationState()函数，输入目标通讯状态：{1,0,0,0,0,0}  即初始状态下，只允许boot报文传输；  调用\*d->initialisation()函数  注意这个case是没有break 的，执行完毕后，继续执行下面的case，即Pre-Operational  所以虽然输入的是Initialisation，但函数执行完后，却是处于Pre-Operational；  **因此程序刚开始调用setState(\*,Initialisation)用于设备初始化，**  **初始化后设备处于Pre-Operational状态；** | | | | | |
| Pre-Operational | | | \*->nodeState=Pre-operational  调用switchCommunicationState()函数，输入目标通讯状态：{0, 1, 1, 1, 1, 0,1}  即预操作状态下，允许：SDO、EMCY、SYNC、HeartBeat、LSS  如果本设备是主机，那么就调用masterSendNMTstateChange()函数，发送NMT命令，初始化所有从机的状态  调用\*d->preOperational()函数 | | | | | |
| Operational | | | \*->nodeState=Operational  调用switchCommunicationState()函数，输入目标通讯状态：{0, 1, 1, 1, 1, 1, 0}  即预操作状态下，允许：SDO、EMCY、SYNC、Heartbeat、PDO  调用\*d->operational()函数 | | | | | |
| Stopped | | | \*->nodeState=Stopped  调用switchCommunicationState()函数，输入目标通讯状态：{0, 0, 0, 0, 1, 0, 1}  即预操作状态下，允许：Heartbeat、LSS  调用\*d->stopped()函数 | | | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| initPreOperationalMode | 输入 | CO\_Data类型指针 | Canopen设备地址 |
| 定义 |  | |

## 配置

|  |  |  |
| --- | --- | --- |
| CO\_Data: | | |
| 成员 | 类型 | 配置 |
| NMT\_Slave\_Node\_Reset\_Callback | void变量 |  |
| NMT\_Slave\_Communications\_Reset\_Callback | void变量 |  |

## 发送NMT报文

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| masterSendNMTstateChange | 输入 | CO\_Data指针 | Canopen设备地址 | |
| UNS8变量 | 报文发送的目标从机设备的Node-ID | |
| UNS8变量 | NMT\_Start\_Node | 控制从机进入运行状态 |
| NMT\_Stop\_Node | 强制设备进入停止状态 |
| NMT\_Enter\_PreOperational | 控制从机进入预操作状态 |
| NMT\_Reset\_Node | 复位从机设备  从机设备接收到该命令后重启，  进入初始化状态的复位应用状态 |
| NMT\_Reset\_Comunication | 复位从机设备  从机设备接收到该命令后重启，  进入初始化状态的复位通信状态 |
| 使用 | 主机发送NMT报文，控制目标从机的状态  调用Cansend函数 | | |

## 接收NMT报文

|  |  |  |  |
| --- | --- | --- | --- |
| proceedNMTstateChange | 输入 | CO\_Data指针 | Canopen设备地址 |
| Message指针 | 创建一个Message变量，存储从机收到的来自主机的NMT报文  输入该Message变量的地址； |
| 使用 | 本函数在canDispatch()函数中被调用，当设备收到CAN报文，就会进入CAN中断，然后执行CAN接收中断服务函数中的canDispatch()函数；  canDispatch()函数中，判断所收到的CAN报文的功能码，  如果功能码是NMT(#define NMT 0x0) ，那么就会执行本函数； | |
| 判断本设备是否是从机：\*->d->iam\_a\_slave  因为只有从机设备才能接收NMT命令，只有从机设备才能执行本函数， | |
| 查看输入的CAN报文的数据场的data[0]，获取所收到的NMT报文的cs标志  代表来自主机的5种NMT命令；  调用setState()函数，设置本设备到对应的节点状态； | |

# NEC

## 配置心跳报文

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| CO\_Data: | | | | |
| 成员 | 类型 | 配置 | | |
| ConsumerHeartbeatCount | UNS8指针 | 1016对象只有一个00 条目，条目内容是一个UNS32变量，  传递该变量的地址，  该条目表示的是1016对象的条目数；  即本设备作为心跳报文Consumer时，producer的数目；  一台设备可以同时接收来自多个设备(生产者)的心跳报文； | | |
| ConsumerHeartbeatEntries | UNS32指针 | 创建一个UNS32类型数组，作为**心跳生产组：**  传递该数组的首地址  数组每一个元素对应一个心跳接收服务器；对应一个心跳报文生产者；对应一个1016对象的条目；  1016对象的条目包含有对应心跳接收服务器的心跳周期以及对应生产者的NodeID，如：  ConsumerHeartbeatEntries[0]对应第0个心跳接收服务器；  对应第0个心跳报文的生产者；  ConsumerHeartbeatEntries[0]的值是**1016-01** 条目的值 | | |
| ConsumerHeartBeatTimers | TIMER\_HANDLE  指针 | 本设备作为Consumer，要为它的每一个心跳接收服务器设置一个定时任务，创建一个TIMER\_HANDLE类型数组，数组元素就是对应的定时任务的状态；如  ConsumerHeartBeatTimers[0]对应第0个心跳接收服务器的定时任务的状态：  传递数组的首地址； | | |
| TIMER\_FREE |  | |
| TIMER\_ARMED |  | |
| TIMER\_TRIG |  | |
| TIMER\_TRIG\_PERIOD |  | |
| TIMER\_NONE |  | |
| ProducerHeartBeatTime | UNS16指针 | 1017对象只有一个00 条目，条目内容是一个UNS32变量，  该条目表示本设备作为心跳报文生产者时的**心跳间隔**，  即每隔多长时间发送一条心跳报文；  0🡪表示本设备不生产心跳报文； | | |
| ProducerHeartBeatTimer | TIMER\_HANDLE  变量 | 直接赋值🡪本设备的心跳生产服务器对应的定时任务的状态  设备只有一个心跳生产服务器；对应一个定时任务  该定时任务的序号是0； | | |
| TIMER\_FREE | | 空闲状态 |
| TIMER\_ARMED | | 工作状态 |
| TIMER\_TRIG | |  |
| TIMER\_TRIG\_PERIOD | |  |
| TIMER\_NONE | |  |
| heartbeatError | heartbeatError\_t  变量 | typedef void (\*heartbeatError\_t)(CO\_Data\*, UNS8);  创建上述类型的函数，传递该函数的地址，即函数名 | | |

## 配置节点保护

|  |  |  |
| --- | --- | --- |
| CO\_Data: | | |
| NMTable[NMT\_MAX\_NODE\_ID] | e\_nodeState  变量 | 创建e\_nodeState类型数组，传递数组的所有元素变量的数值  数组容量：NMT\_MAX\_NODE\_ID，定义位置：config.h  这是主机的**从机监视表**；  主机通过节点保护报文获取从机设备的状态后，就会将状态存储到该数组对应的元素中；  数组元素的索引号就是对应从机的NodeID  如NMTable[0x71] 🡪用于存储NodeID=1110001 的设备的节点状态 |
| post\_SlaveBootup | post\_SlaveBootup\_t  变量 | typedef void (\*post\_SlaveBootup\_t)(CO\_Data\*, UNS8);  创建上述类型的函数，传递该函数的地址，即函数名 |
| post\_SlaveStateChange | post\_SlaveStateChange\_t  变量 | typedef void (\*post\_SlaveStateChange\_t)(CO\_Data\*, UNS8, e\_nodeState);  创建上述类型的函数，传递该函数的地址，即函数名 |
| toggle | UNS8变量 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| masterRequestNodeState | 输入 | CO\_Data指针 | Canopen设备地址 |
| UNS8变量 | 某个从机设备的7位Node-ID |
| 使用 | 一般是本设备作为主机，调用该函数，用于配置主机的从机监视表  输入某个从机设备的NodeID，  将对应的NMTable[某个从机设备的7位Node-ID]清0  如果输入的NodeID=0，那么就将整个从机监视表清0 | |

## 发送NEC报文

|  |  |  |  |
| --- | --- | --- | --- |
| ProducerHearbeatAlarm | 输入 | CO\_Data指针 | Canopen设备地址 |
| UNS32变量 | 心跳服务器的序号 |
| 使用 | 本设备发送心跳报文，报文的数据场是本设备的节点状态 \*.nodeState | |
| slaveSendBootUp | 输入 | CO\_Data指针 | Canopen设备地址 |
| 使用 | 本设备发送Boot报文  无论什么设备，一旦从初始化状态进入与操作状态，就会自动发布启动报文，  设备将自己的这一状态改变事件（即设备从初始化状态转变到预操作状态这一事件）通知其他设备。 | |
| masterSendNMTnodeguard | 输入 | CO\_Data指针 | Canopen设备地址 |
| UNS8变量 | 目标设备的7位Node-ID |
| 使用 | 本设备作为主机，发送节点监视报文到目标从机设备；  该报文是远程帧报文，即rtr=1；  数据场为空，即DLC=0  如果目标从机开启了节点保护功能，那么收到该条报文后，就会回复一条节点保护报文； | |

|  |  |  |  |
| --- | --- | --- | --- |
| post\_SlaveStateChange\_t | 输入 | CO\_Data  指针 | Canopen设备地址 |
| UNS8  变量 | heartbeatID |
| e\_nodeState  变量 |  |
| 使用 | 函数地址定义 | |

|  |  |  |  |
| --- | --- | --- | --- |
| post\_SlaveBootup\_t | 输入 | CO\_Data指针 | Canopen设备地址 |
| UNS8变量 | heartbeatID |
| 使用 |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 创建  heartbeatError\_t  类型函数 | 输入 | CO\_Data指针 | Canopen设备地址 |
| UNS8变量 | 心跳接收服务器的序号 |
| 使用 |  |  |

## 接收NEC报文

|  |  |  |  |
| --- | --- | --- | --- |
| ConsumerHearbeatAlarm | 输入 | CO\_Data指针 | Canopen设备地址 |
| UNS32变量 | 心跳接收服务器的序号 |
| 使用 | 本设备接收心跳报文，如果超过心跳周期消费者还是没有收到指定设备的心跳报文，就会调用本函数；  🡪配置该心跳接收服务器对应的定时任务的状态  🡪配置本函数的从机监视表  🡪调用本设备的心跳错误函数，即\*.heartbeatError | |
| proceedNODE\_GUARD | 输入 | CO\_Data指针 | 本设备(CO\_Data变量)地址 |
| Message指针 | 创建一个Message变量，存储该设备收到的NEC报文，即节点监视报文  输入该Message变量的地址； |
| 使用 | 本函数在canDispatch()函数中被调用，当设备收到CAN报文，就会进入CAN中断，然后执行CAN接收中断服务函数中的canDispatch()函数；  canDispatch()函数中，判断所收到的CAN报文的功能码，  如果功能码是NODE\_GUARD(#define NODE\_GUARD 0xE) ，那么就会执行本函数； | |
| 查看该节点监视报文的Node-ID，看看究竟是不是发送给本设备的；  如果确实是发送给本设备的，那么回复节点保护报文🡪节点监视报文的发送方(主机) | |
|  | |

## 启动传输

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| heartbeatInit  启动 | 输入 | | CO\_Data指针 | | Canopen设备地址 | |
| 使用 | | 启动心跳报文  接收功能 | | 遍历本设备的心跳生产组；查看每个心跳接收服务器的心跳周期  如果心跳周期大于0，那么启动该心跳接收服务器：  调用SetAlarm函数，启动该心跳接收服务器的定时任务：  输入该心跳服务器的序号作为定时任务的序号，  输入ConsumerHearbeatAlarm()函数的地址，  输入心跳周期作为定时时间长度 | |
| 正常情况下，本设备一旦收到来自某个生产者的心跳报文，那么该生产者对应的心跳接收服务器 对应的定时任务重新启动；  如果该定时任务溢出，意味着：  经过设置的定时时间长度(心跳周期)，本设备仍然没有收到心跳报文，那么就认为该生产者发生错误，调用ConsumerHearbeatAlarm()函数 | |
| 启动心跳报文  发布功能 | | 查看本设备的心跳间隔，即 \*.ProducerHeartBeatTime  如果心跳间隔大于0，那么启动本设备的心跳发送服务器：  调用SetAlarm函数，启动心跳发送服务器的定时任务：  输入0作为定时任务的序号，  输入ProducerHearbeatAlarm()函数的地址，  输入心跳间隔作为定时时间长度 | |
| 当该定时任务溢出，即：经过设置的定时时间长度(心跳间隔)，  说明距离上次发送心跳报文已经经过了心跳间隔的时间长度，那么  调用ProducerHearbeatAlarm()函数再次发送心跳报文；  然后重新启动心跳发送服务器的定时任务 | |
| heartbeatStop  停止 | 输入 | | CO\_Data指针 | | Canopen设备地址 | |
| 使用 | | 遍历本设备的心跳生产组；对于每一个心跳接收服务器，都使用DelAlarm()函数，停止其对应的定时任务； | | | |
| 对本设备的心跳生产服务器，使用DelAlarm()函数，停止其对应的定时任务； | | | |
|  | | | | | | |
| OnHeartbeatProducerUpdate  重新启动 | | 输入 | | CO\_Data指针 | | Canopen设备地址 |
| indextable指针 | |  |
| UNS8指针 | |  |
| 使用 | | 先调用heartbeatStop()函数，再调用heartbeatInit()函数 | | |

# EMCY

## 应急错误

|  |  |  |  |
| --- | --- | --- | --- |
| s\_error | 成员 | 类型 | 说明 |
| errCode | UNS16 | 该错误的16位应急错误代码的值 |
| errRegMask | UNS8 | 该错误对应的错误寄存器的值 |
| active | UNS8 | 1：发生该错误  0：该错误没有发生 |
| 一个变量s\_error表示一个应急错误 | | |

|  |  |  |  |
| --- | --- | --- | --- |
| post\_emcy\_t | 输入 | CO\_Data指针 |  |
| UNS8变量 |  |
| UNS16变量 |  |
| UNS8变量 |  |
| 这是一个数据类型，用来定义一个函数的指针 | | |
| \_post\_emcy |  | | |

|  |  |  |  |
| --- | --- | --- | --- |
| CO\_Data: | | | |
| error\_state | e\_errorState  变量 | Error\_free | 当前本设备正常 |
| Error\_occurred | 当前本设备处于错误状态 |
| error\_history\_size | UNS8变量 | 当前本设备中错误的个数 | |
| error\_number | UNS8指针 | 1003对象的00 条目，条目内容是一个UNS8变量，  传递该变量的地址，  该条目表示当前1003对象中的最高子索引值 | |
| error\_first\_element | UNS32指针 | 1003对象的01 条目，条目内容是一个UNS32变量，  传递该变量的地址，  该条目表示最新的错误的信息 | |
| error\_register | UNS8指针 | 1000对象只有一个00 条目，条目内容是一个UNS8变量，  传递该变量的地址  该条目是本设备的**8位错误寄存器** | |
| error\_cobid | UNS32指针 | 1014对象只有一个00 条目，条目内容是一个UNS32变量，  传递该变量的地址，  配置bit30：1🡪使能EMCY报文发送功能，0🡪禁止发送EMCY报文  配置低11位，这是本设备发送的SYNC报文的CAN-ID  其余位全部为0 | |
| error\_data[EMCY\_MAX\_ERRORS] | s\_errors变量 | 创建s\_errors类型数组，传递整个数组作为该成员  数组容量：EMCY\_MAX\_ERRORS，定义位置：emcy.h  这是该设备的应急错误表； | |
| post\_emcy | post\_emcy\_t  变量 | 创建post\_emcy\_t类型的函数，  传递该函数的地址，即函数名 | |

## 发送EMCY报文

|  |  |  |  |
| --- | --- | --- | --- |
| sendEMCY | 输入 | CO\_Data指针 | Canopen设备地址 |
| UNS16变量 | 输入16位应急错误代码 |
| UNS8变量 | 输入错误寄存器的值 |
| 使用 | 调用canSend()函数发送EMCY报文  报文的CANID使用的是\*->error\_cobid，  报文数据场长度是8  数据场内容：  data[0]：16位应急错误代码的低8位  data[1]：16位应急错误代码的高8位  data[2]：错误寄存器的值  data[3~7]：待定 | |
| EMCY\_setError | 输入 | CO\_Data指针 | Canopen设备地址 |
| UNS16变量 | 输入 该应急错误的 16位应急错误代码 |
| UNS8变量 | 输入 该应急错误的 对应的错误寄存器的值 |
| UNS16变量 | 输入 该应急错误的 16位附加信息 |
| 输出 | UNS8变量 |  |
| 使用 | 当该设备发生错误时，就会调用本函数，  输入所发生的的错误对应的错误代码以及错误寄存器值，执行本函数如下： | |
| 配置  应急错误 | 遍历该设备的**应急错误表**，直到找到一个空闲的应急错误(active=0)，  按照输入的参数配置该应急错误；即配置其对应的s\_error变量的各个成员 |
| 配置对象 |  |
| 发送EMCY | 调用sendEMCY()函数，发送该应急错误 |
| EMCY\_errorRecovered | 输入 | CO\_Data指针 | Canopen设备地址 |
| UNS16变量 | 输入 该应急错误的 16位应急错误代码 |
| 使用 | 当设备的某个错误恢复时，就会调用本函数  输入所发生的的错误对应的错误代码，执行本函数如下： | |
| 寻找  应急错误 | 遍历该设备的**应急错误表**，根据所输入的应急错误代码，  找到该应急错误在应急错误表中的索引index  那么这个恢复的错误就是error\_data[index] |
| 配置对象 |  |
| 发送EMCY | 调用sendEMCY()函数，  发送EMCY报文(错误代码为0x00xx，表示错误复位) |

## 接收EMCY报文

|  |  |  |  |
| --- | --- | --- | --- |
| proceedEMCY | 输入 | CO\_Data指针 | Canopen设备地址 |
| Message指针 | 创建一个Message变量，存储本设备收到的来自其他设备的EMCY报文  输入该Message变量的地址； |
| 使用 | 本函数在canDispatch()函数中被调用，当设备收到CAN报文，就会进入CAN中断，然后执行CAN接收中断服务函数中的canDispatch()函数；  canDispatch()函数中，判断所收到的CAN报文的功能码，  如果功能码是SYNC(#define SYNC 0x1 ，EMCY和SYNC共用) ，并且查看设备此刻的通讯状态:  EMCY处于启动状态：\*->CurrentCommunicationState.csEmergency=1  那就执行本函数 | |
| 调用 \*->post\_emcy  做出相应的处理 | |

## 启动传输

|  |  |  |  |
| --- | --- | --- | --- |
| emergencyInit | 输入 | CO\_Data指针 | Canopen设备地址 |
| 使用 | 本设备启动EMCY发送  调用RegisterSetODentryCallBack()函数  设置\*->error\_number = 0，因为在初始情况下，设备不存在错误 | |
| emergencyStop | 输入 | CO\_Data指针 | Canopen设备地址 |
| 使用 | 本设备禁止EMCY发送  调用RegisterSetODentryCallBack()函数 | |

|  |  |  |  |
| --- | --- | --- | --- |
| OnNumberOfErrorsUpdate | 输入 | CO\_Data指针 | Canopen设备地址 |
| indextable指针 |  |
| UNS8变量 |  |
| 输出 | UNS32变量 |  |
| 使用 |  | |

# SDO

## 创建SDO服务器

|  |  |  |
| --- | --- | --- |
| CO\_Data: | | |
| 成员 | 类型 | 说明 |
| transfers[SDO\_MAX\_SIMULTANEOUS\_TRANSFERS] | s\_transfer transfers  变量 | 数组容量：SDO\_MAX\_SIMULTANEOUS\_TRANSFERS  位置：config.h ，范围：[1,256] |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 一个s\_transfers变量表示一个SDO服务器，SDO服务器分为两种：  “本设备作为客户端”的s\_transfers变量我称之为客户型服务器；  “本设备作为服务端”的s\_transfers变量我称之为服务型服务器；  客户型服务器的部分内容是根据对应的SDO client parameter对象 配置的  服务型服务器的部分内容是根据对应的SDO server parameter对象 配置的  一个CANopen设备共配置 (SDO\_MAX\_SIMULTANEOUS\_TRANSFERS+1) 个SDO服务器；  一个CANopen设备至少有1个SDO服务器，即SDO1， | | | | |
| 在某一段时间里，设备使用它的某个SDO服务器进行SDO传输，那么：  根据该设备在此次SDO传输中扮演的角色，配置其对应的s\_transfers变量(客户型服务器或服务型服务器)，  该s\_transfers变量在**一次SDO传输**中：  有些成员保持不变：如whoami、CliServNbr、等，这些成员是一次**SDO传输**的固有属性；  有些成员则一直在变化：如toggle、index、subindex、等，因为一次这些成员一次SDO传输包含有多个SDO报文，  这些成员表示的是该SDO服务器在本次SDO传输中**传输的某个SDO报文的属性**； | | | | |
| s\_transfers： | | | | |
| 成员 | 类型 |  | | |
| CliServNbr | UNS8变量 | SDO服务器的序号 | | |
| whoami | UNS8变量 | SDO\_SERVER | 在本次SDO传输中，本设备作为服务端 | |
| SDO\_CLIENT | 在本次SDO传输中，本设备作为客户端 | |
| SDO\_UNKNOWN | 不确定 | |
| state | UNS8变量 | SDO\_RESET | | 该SDO服务器的状态：空闲 |
| SDO\_FINISHED | | SDO传输完毕 |
| SDO\_ABORTED\_RCV | | 此刻正处于中断传输过程中 |
| SDO\_ABORTED\_INTERNAL | | 此刻正处于中断传输过程中 |
| SDO\_DOWNLOAD\_IN\_PROGRESS | | 此刻正处于**快速/分段下载**传输过程中 |
| SDO\_UPLOAD\_IN\_PROGRESS | | 此刻正处于**快速/分段上传**传输过程中 |
| SDO\_BLOCK\_DOWNLOAD\_IN\_PROGRESS | | 此刻正处于**块下载**传输过程中 |
| SDO\_BLOCK\_UPLOAD\_IN\_PROGRESS | | 此刻正处于**块上传**传输过程中 |
| toggle | UNS8变量 | 触发位； 取值0或1 ； 由ProceedSDO()函数更新；  当该SDO服务器要**发送**一条片段报文(下载片段/上传片段/回复片段)，配置该报文时，  该片段报文的toggle位就使用该SDO服务器的toggle成员的当前的值； | | |
| 每当该SDO服务器发送完毕一条片段报文，就将其触发位数值反转；  该数值就是下一次发送片端报文时使用的触发位数值 | | |
| abortCode | UNS32变量 | 仅当state = SDO\_ABORTED\_INTERNAL时有效；  如果该SDO服务器发送了一条SDO中断报文，那么意味着本次SDO传输的结束，子服务器进入SDO\_ABORTED\_INTERNAL状态；在复位之前，该子服务器一直保持在该状态，  其abortCode存储了该SDO中断报文的中断码 | | |
| index | UNS16变量 | 本次SDO传输的索引号数值 | | |
| subIndex | UNS8变量 | 本次SDO传输的子索引号数值 | | |

|  |  |  |  |
| --- | --- | --- | --- |
| count | UNS32变量 | 本次读取或写入数据暂存区的数据的字节数 | |
| offset | UNS32变量 | 该SDO服务器的数据暂存区的指针，每次读取data[]中的数据或者是将数据写入到data[]中，offset都会做相应的移动：  当写入数据到data []中时，data[offset]是下一个要写入的位置；  当从data []中读取数据时，data[offset]是下一个要读取的字节； | |
| data [] | UNS8变量 | data[SDO\_MAX\_LENGTH\_TRANSFER] 🡪 该SDO服务器的数据暂存区  该数组存储了一次SDO传输中传输的所有**数据场有效数据** | |
| 如果本次SDO传输中，该SDO服务器执行数据发送，那么在SDO传输之前，就将所有要发送的数据都预先存储在了data[]中，每次要发送一个SDO下载报文，都会从data[]中提取一定字节数的数据作为该SDO报文数据场的有效字节，然后offset做相应的移动； | |
| 如果本次SDO传输中，该SDO服务器执行数据接收，那么在SDO传输之前，就会将data[]清空，每次接收一个SDO报文，都会从该SDO报中提取出有效数据，存储在data[]中，然后offset做相应的移动  本次SDO传输完毕后，才会处理data[]中的数据，即调用SDOlineToObjdict()函数，将数据传递到指定的对象条目中； | |
| dynamicData | UNS8指针 |  | |
| dynamicDataSize | UNS32变量 |  | |
| peerCRCsupport | UNS8变量 | 1：支持CRC  0：不支持CRC | |
| blksize | UNS8变量 | 表示SDO块传输时该SDO报文包含的片段的数目  范围：[1,127] | |
| ackseq | UNS8变量 | 最后一个成功接收到的SDO片段的序列号 | |
| objsize | UNS32变量 |  | |
| lastblockoffset | UNS8变量 |  | |
| seqno | UNS8变量 | 最后一个成功接收或发送的序列号 | |
| endfield | UNS8变量 | 最后一个数据块的最后一个片段中的无效字节数 | |
| rxstep | rxStep\_t变量 | RXSTEP\_INIT |  |
| RXSTEP\_STARTED |  |
| RXSTEP\_END |  |
| tmpData[8] | UNS8变量 | 用于储存临时的片段 | |
| dataType | UNS8变量 |  | |
| timer | TIMER\_HANDLE  变量 | 该SDO服务器的计数器 | |
| Callback | SDOCallback\_t  变量 | 回调函数的地址，即回调函数的函数名  每一个自服务期对应一个SDO回调函数 | |

## 配置SDO服务器

|  |  |  |  |
| --- | --- | --- | --- |
| resetSDO | 输入 | CO\_Data指针 | Canopen设备地址 |
| 定义 | 复位该设备的所有的s\_transfers变量，调用resetSDOline()函数 | |
| resetSDOline | 输入 | CO\_Data指针 | Canopen设备地址 |
| UNS8变量 | SDO服务器序号 |
| 定义 | 复位某一个SDO服务器：  清零 data []、whoami、abortCode 这3个成员；  调用initSDOline()函数； | |
| initSDOline | 输入 | CO\_Data指针 | Canopen设备地址 |
| UNS8变量 | SDO服务器序号 |
| UNS8变量 | 设置该SDO服务器的参数🡪CliServNbr |
| UNS16变量 | 设置该SDO服务器的参数🡪index |
| UNS8变量 | 设置该SDO服务器的参数🡪subIndex |
| UNS8变量 | 设置该SDO服务器的参数🡪state |
| 定义 | 初始化某一个SDO服务器  🡪按照输入的参数的值来设置 CliServNbr、index、subIndex、state 这4个成员  🡪调用StartSDO\_TIMER()或StopSDO\_TIMER() 设置timer成员  🡪rxstep成员设置为RXSTEP\_INIT  🡪除了由resetSDOline()函数清0的3个成员、本函数设置的6个成员以外，清零所有其他成员 | |
| StartSDO\_TIMER | 输入 | SDO服务器序号 | |
| 使用 | 专门用来设置某个SDO服务器的timer成员：  调用Set\_Alarm函数，为该SDO服务器创建一个定时器 | |
| StopSDO\_TIMER | 输入 | SDO服务器序号 | |
| 使用 | 专门用来设置某个SDO服务器的timer成员：  调用Del\_Alarm函数，删除该SDO服务器的定时器 | |
| RestartSDO\_TIMER | 输入 | SDO服务器序号 | |
| 使用 | 专门用来设置某个SDO服务器的timer成员：  先调用Del\_Alarm函数，再调用Set\_Alarm函数：重启该子服务器的定时器 | |
| getSDOlineRestBytes | 输入 | CO\_Data指针 | Canopen设备地址 |
| UNS8变量 | SDO服务器序号 |
| UNS32指针 | 预先创建一个空白的UNS32变量，将该变量的地址输入  该变量将用来存储该SDO服务器的count值 |
| 输出 | UNS8变量 |  |
| 使用 |  | |
| setSDOlineRestBytes | 输入 | CO\_Data指针 | Canopen设备地址 |
| UNS8变量 | SDO服务器序号 |
| UNS32变量 | 预先创建一个UNS32变量，赋值要设置的count的数值  将该变量的地址输入 |
| 输出 | UNS8变量 |  |
| 使用 |  | |

|  |  |  |  |
| --- | --- | --- | --- |
| SDOTimeoutAlarm | 输入 | CO\_Data指针 | Canopen设备地址 |
| UNS32变量 | SDO服务器序号 |
| 使用 | 该Canopen设备作为SDO传输的客户端，**发送**SDO报文时，使用本函数 | |

## 查看SDO服务器

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 大多数情况下，一个CANopen设备设置了多个SDO服务器，  这些SDO服务器的配置各不相同，所处的状态也不相同；  当设备要进行SDO传输时，需要选择合适的SDO服务器；  用户可以使用下面这些函数，输入需求，函数就会找到合适的SDO服务器 | | | | | | | |
| GetSDOClientFromNodeId | 输入 | CO\_Data指针 | Canopen设备地址 | | | | |
| UNS8变量 | 某个SDO传输中对方设备(服务端设备)的Node-ID | | | | |
| 输出 | UNS8变量 | 0xFF | | | | 没有配置SDO client parameter |
| 0xFE | | | | 没有找到匹配的SDO服务器 |
| 其它 | | | | 本设备所使用的SDO服务器的序号 |
| 使用 | 从第一个SDO服务器开始，遍历所有的SDO client parameter对象的条目4  一旦发现**和输入的NodeID一致**，就会输出本设备所使用的SDO服务器的序号； | | | | | |
| 在某个SDO传输中，本设备作为SDO客户端，现已知对方设备(即服务端的NodeID)  通过本函数可以获取在本次SDO传输中本设备所使用的SDO服务器的序号； | | | | | |
| 要了解的是某个设备的SDOn client parameter对象的第4个条目(对应子索引号=03)  该条目表示本设备使用该SDO服务器，作为客户端进行SDO传输时，对方设备(即服务端设备)的NodeID的数值； | | | | | |
| getSDOfreeLine | 输入 | CO\_Data指针 | | Canopen设备地址 | | | |
| UNS8变量 | | SDO\_SERVER | | 如果查找成功，配置该空闲的SDO服务器的whoami成员 | |
| SDO\_CLIENT | |
| UNS8指针 | | 事先创建一个空白的UNS8变量，将该变量的地址输入  该变量将用来存储本函数找到的空闲SDO服务器的序号值 | | | |
| 输出 | UNS8变量 | | 0 | | 查找成功， | |
| 0xFF | | 查找失败，表示所有的SDO服务器均处于工作状态 | |
| 使用 | 从第一个SDO服务器开始，遍历该CANopen设备的所有SDO服务器，  直到发现一个**空闲的**SDO服务器(state=SDO\_RESET )； | | | | | |
| getSDOlineOnUse | 输入 | CO\_Data指针 | | | Canopen设备地址 | | |
| UNS8变量 | | | 指定要查看的SDO服务器的序号(CliServNbr成员的值) | | |
| UNS8变量 | | | SDO\_SERVER | | 指定该SDO服务器此刻正作为服务端 |
| SDO\_CLIENT | | 指定该SDO服务器此刻正作为客户端 |
| UNS8指针 | | | 事先创建一个空白的UNS8变量，将该变量的地址输入  该变量将用来存储本函数找到的空闲SDO服务器的序号值 | | |
| 输出 | UNS8变量 | | | 0 | | 该SDO服务器处于快速传输或分段传输或块传输状态，并且，处于用户指定的服务中； |
| 0xFF | | 该SDO服务器处于空闲状态，或，处于中断传输状态，  或，与用户指定的服务不一致； |
| getSDOlineToClose | 输入 | CO\_Data指针 | | | Canopen设备地址 | | |
| UNS8变量 | | | 指定要查看的SDO服务器的序号(CliServNbr成员的值) | | |
| UNS8变量 | | | SDO\_SERVER | | 指定该SDO服务器此刻正作为服务端 |
| SDO\_CLIENT | | 指定该SDO服务器此刻正作为客户端 |
| UNS8指针 | | | 事先创建一个空白的UNS8变量，将该变量的地址输入  该变量将用来存储本函数找到的空闲SDO服务器的序号值 | | |
| 输出 | UNS8变量 | | | 0 | | 该SDO服务器处于非空闲状态，  并且，处于用户指定的服务中； |
| 0xFF | | 该SDO服务器处于空闲状态，  或，与用户指定的服务不一致； |

## 发送SDO报文

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| sendSDO | 输入 | CO\_Data指针 | | 本设备对应的CO\_DATA变量的地址 | | | | |
| UNS8变量 | | SDO\_SERVER | | | 本次SDO发送中，本设备作为服务端，  该SDO报文传输方向：服务端🡪客户端 | |
| SDO\_CLIENT | | | 本次SDO发送中，本设备作为客户端，  该SDO报文传输方向：客户端🡪服务端 | |
| UNS8变量 | | 本次SDO发送所使用的SDO服务器的序号 | | | | |
| UNS8指针 | | 预先创建一个UNS8数组，数组容量是8，存储SDO报文的数据场；  输入数组的首地址； | | | | |
| 使用 | 调用canSend函数发送SDO报文  该SDO报文的数据场由函数输入的UNS8类型数组确定；  该SDO报文的其他参数(如CAN-ID、DLC等)则由函数选择的SDO服务器的配置来确定； | | | | | | |
| sendSDOabort | 输入 | CO\_Data指针 | | 本设备对应的CO\_DATA变量的地址 | | | | |
| UNS8变量 | | SDO\_SERVER | | 本次SDO发送中，本设备作为服务端，  该SDO报文传输方向：服务端🡪客户端 | | |
| SDO\_CLIENT | | 本次SDO发送中，本设备作为客户端，  该SDO报文传输方向：客户端🡪服务端 | | |
| UNS8变量 | | 本次SDO发送所使用的SDO服务器的序号 | | | | |
| UNS16变量 | | 该中断报文的数据场的byte1~2的内容，即索引数值 | | | | |
| UNS8变量 | | 该中断报文的数据场的byte3的内容，即子索引数值 | | | | |
| UNS32变量 | | 该中断报文的数据场的byte4~7的内容，即中断码的数值 | | | | |
| 使用 | 调用sendSDO函数发送SDO中断报文  该SDO报文的数据场由函数的最后3个输入参数确定；  该SDO报文的其他参数(如CAN-ID、DLC等)则由函数选择的SDO服务器的配置来确定； | | | | | | |
| SDOlineToObjdict | 输入 | CO\_Data指针 | | Canopen设备地址 | | | | |
| UNS8变量 | | 本次SDO发送所使用的SDO服务器的序号 | | | | |
| 输出 | UNS32变量 | | 0 | | | | 写入成功 |
| 0xFF | | | | 操作失败 |
| 使用 | 一般是该SDO服务器在某次SDO传输中接收数据，它将整次SDO传输中接收的所有数据都存放在它的data[]成员中；  传输结束后调用本函数，将data[]中的数据传递到本次SDO传输对应的对象条目中(即该SDO服务器的index成员和subindex成员) | | | | | | |
| objdictToSDOline | 输入 | CO\_Data指针 | | Canopen设备地址 | | | | |
| UNS8变量 | | 本次SDO发送所使用的SDO服务器的序号 | | | | |
| 输出 | UNS32变量 | | 0 | | | | 写入成功 |
| 0xFF | | | | 操作失败 |
| 使用 | 一般是该SDO服务器在某次SDO传输中发送数据，  即作为客户端发送一个SDO下载请求 或 作为服务端收到一个上传请求；  这时会调用本函数，该SDO服务器知道在本次SDO传输中要发送哪些数据(找到相应的对象条目)，它将这些数据预先全部存储在其data[]成员中；  接下来再执行发送； | | | | | | |
| SDOtoLine | 输入 | CO\_Data指针 | Canopen设备地址 | | | | | |
| UNS8变量 | SDO服务器的序号 | | | | | |
| UNS32变量 | 指定要写入的字节数 | | | | | |
| UNS8指针 | 预先创建一个UNS8类型数组，该数组存储要写入的数据；输入该数组的首地址 | | | | | |
| 输出 | UNS8变量 | 0 | | 写入成功 | | | |
| 0xFF | | 操作失败 | | | |
| 使用 | 写入数据到指定的SDO服务器的数据暂存区(即data[]成员)  与objdictToSDOline()函数有着相同的目的：都是将数据写入到子服务器的数据暂存区中；  但本函数相对比较自由，可以任意选择要存储的数据，而不是只能从对象条目中提取； | | | | | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| lineToSDO | 输入 | CO\_Data指针 | Canopen设备地址 | |
| UNS8变量 | SDO服务器的序号 | |
| UNS32变量 | 指定要读取的字节数 | |
| UNS8指针 | 预先创建一个空白UNS8类型数组，输入该数组的首地址  本函数在执行过程中将指定要获取的数据存储在该数组中； | |
| 输出 | UNS8变量 | 0 | 读取成功 |
| 0xFF | 操作失败 |
| 使用 | 从SDO服务器的数据暂存区(即data[]成员)读取数据  写入数据到指定的SDO服务器的数据暂存区(即data[]成员)  与SDOlineToObjdict()函数有着相同的目的：都是子服务器的数据暂存区中读取数据；  但本函数相对比较自由，可以选择读取数据的处理方式，而不是只能传递到对象条目中； | | |

## SDO请求

根据CANopen协议我们知道发送SDO请求的设备称为SDO客户端

**发送SDO请求即发起一次SDO传输**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| SDO  下载  请求 | \_writeNetworkDict  发送下载请求报文 | 输入 | CO\_Data指针 | Canopen设备地址 | |
| UNS8变量 | 本次SDO传输的对方设备(即服务端设备)的NodeID | |
| UNS16变量 | 该SDO请求报文的数据场的byte1~2的内容，即索引数值 | |
| UNS8变量 | 该SDO请求报文的数据场的byte3的内容，即子索引数值 | |
| UNS32变量 | 该SDO请求报文的有效字节数 | |
| UNS8变量 | 数据类型 | |
| void指针 | 要写入的数据内容 | |
| SDOCallback\_t  变量 | 创建一个SDO回调函数  传递该函数的地址(函数名) | |
| UNS8变量 | 0：数据类型  1: | |
| UNS8变量 | 0：块传输  1：快速传输/分段传输 | |
| 输出 | UNS8变量 | 0 | 发送成功 |
| 0xFF | 错误 |
| 0xFE | 错误 |
| writeNetworkDict | 调用\_writeNetworkDict 函数，选定快速传输/分段传输，Callback=NULL， | | | |
| writeNetworkDictCallBack | 调用\_writeNetworkDict 函数，选定快速传输/分段传输， | | | |
| writeNetworkDictCallBackAI | 调用\_writeNetworkDict 函数， | | | |
| getWriteResultNetworkDict | 输入 | CO\_Data指针 | Canopen设备地址 | |
| UNS8变量 |  | |
| UNS32指针 |  | |
| 输出 | UNS8变量 |  | |
| 使用 |  | | |
| SDO  上传  请求 | \_readNetworkDict  发送上传请求报文 | 输入 | CO\_Data指针 | Canopen设备地址 | |
| UNS8变量 | 本次SDO传输的对方设备(即服务端设备)的NodeID | |
| UNS16变量 | 该SDO请求报文的数据场的byte1~2的内容，即索引数值 | |
| UNS8变量 | 该SDO请求报文的数据场的byte3的内容，即子索引数值 | |
| UNS8变量 | 数据类型 | |
| SDOCallback\_t  变量 |  | |
| UNS8变量 | 0：块传输  1：快速传输/分段传输 | |
| 输出 | UNS8变量 |  | |
| readNetworkDict | 调用\_readNetworkDict，Callback=NULL | | | |
| readNetworkDictCallback | 调用\_readNetworkDict，两个函数完全相等 | | | |
| readNetworkDictCallbackAI | 调用\_readNetworkDict， | | | |
| getReadResultNetworkDict | 输入 | CO\_Data指针 | Canopen设备地址 | |
| UNS8变量 |  | |
| void指针 |  | |
| UNS32指针 |  | |
| 输出 | UNS8变量 |  | |
| 使用 |  | | |

## SDO传输

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| proceedSDO | 输入 | CO\_Data指针 | Canopen设备地址 | |
| Message指针 | 创建一个Message变量，存储从机收到的来自主机的SDO报文  输入该Message变量的地址； | |
| 输出 | UNS8变量 |  | |
| 使用 | 本函数在canDispatch()函数中被调用，当设备收到CAN报文，就会进入CAN中断，然后执行CAN接收中断服务函数中的canDispatch()函数；  canDispatch()函数中，判断所收到的CAN报文的功能码，  如果功能码是SDOtx(#define SDOtx 0xB) 或 SDOrx(#define SDOrx 0xC )，那么就会执行本函数； | | |
| 当设备收到SDO报文时，提取该报文，即Message类型变量；  输入该变量地址给proceedSDO()函数，函数根据输入的Message变量来发送相应回复报文  每当设备收到CAN报文都会触发中断，接着提取收到的报文内容，然后执行proceedSDO()函数 | | |
| failedSDO | 输入 | CO\_Data指针 | Canopen设备地址 | |
| UNS8变量 | 本次SDO传输所使用的SDO服务器的序号 | |
| UNS8变量 | SDO\_SERVER | 本次SDO发送中，本设备作为服务端， |
| SDO\_CLIENT | 本次SDO发送中，本设备作为客户端， |
| UNS16变量 | 发送的中断报文的数据场的byte1~2的内容，即索引数值 | |
| UNS8变量 | 发送的中断报文的数据场的byte3的内容，即子索引数值 | |
| UNS32变量 | 发送的中断报文的数据场的byte4的内容，即中断码的数值 | |
| 输出 | UNS8变量 |  | |
| 使用 | 本函数在proceedSDO()函数中被调用；用于终止本次SDO传输  因为在SDO传输中，总是在本设备收到了一条报文，使用proceedSDO()函数处理该报文的进程中发现收到的报文不正确，然后调用本函数终止本次SDO传输； | | |
| 调用resetSDOline()函数复位本次SDO传输所使用的SDO服务器； | | |
| 调用StopSDO\_TIMER()删除该SDO服务器定时器  设置state成员为SDO\_ABORTED\_INTERNAL  更新abortcode成员的数值位本函数输入的中断码的数值 | | |
| 调用sendSDOabort()函数发送SDO中断报文； | | |
| closeSDOtransfer | 输入 | CO\_Data指针 | Canopen设备地址 | |
| UNS8变量 |  | |
| UNS8变量 |  | |
| 输出 | UNS8变量 |  | |
| 使用 |  | | |

# SYNC

## 配置

|  |  |  |  |
| --- | --- | --- | --- |
| CO\_Data: | | | |
| 成员 | 类型 | 配置 | |
| syncTimer | TIMER\_HANDLE  变量 |  | |
| TIMER\_FREE |  |
| TIMER\_ARMED |  |
| TIMER\_TRIG |  |
| TIMER\_TRIG\_PERIOD |  |
| TIMER\_NONE |  |
| COB\_ID\_Sync | UNS32指针 | 1005对象只有一个00 条目，条目内容是一个UNS32变量，传递该变量的地址  配置bit30：1🡪使能SYNC报文发送功能， 0🡪禁止发送SYNC报文  配置低11位，这是本设备发送的SYNC报文的CAN-ID(一般配置为0x80)  其余位全部为0， | |
| Sync\_Cycle\_Period | UNS32指针 | 1006对象只有一个00 条目，条目内容是一个UNS32变量，传递该变量的地址  该条目表示本设备发送SYNC报文的循环周期，即每经过一个循环周期所指定的时间发送一个同步报文；  0：不使用同步传输  1~~0xFFFF FFFF：循环周期的时间值 ，单位：us | |
| Sync\_window\_length | UNS32指针 | 1007对象只有一个00 条目，条目内容是一个UNS32变量，传递该变量的地址  该条目表示本设备接收SYNC报文时的同步窗口长度；  本设备作为PDO生产者，收到指定个数的同步报文后，需要在同步窗口长度指定的时间内发送PDO；  0：发送PDO没有时间限制  1~~0xFFFF FFFF：同步窗口长度的时间值，单位：us  同步窗口长度的数值不能超过循环周期；一般就设置同步窗口长度=循环周期； | |
| post\_sync | post\_sync\_t变量 | typedef void (\*post\_sync\_t)(CO\_Data\*);  创建上述类型的函数，传递函数地址，即函数名 | |

## 发送SYNC报文

|  |  |  |  |
| --- | --- | --- | --- |
| sendSYNCMessage | 输入 | CO\_Data指针 | Canopen设备地址 |
| 输出 | UNS8变量 |  |
| 使用 | 调用canSend()函数发送SYNC报文  报文的CAN-ID使用的是\*->COB\_ID\_Sync，  数据场空 | |
| SyncAlarm | 输入 | CO\_Data指针 | Canopen设备地址 |
| UNS32变量 |  |
| 使用 | **=sendSYNC()** | |
| sendSYNC | 输入 | CO\_Data指针 | Canopen设备地址 |
| 输出 | UNS8变量 |  |
| 使用 | 先调用sendSYNCMessage()函数，  再调用proceedSYNC()函数 | |

## 启动传输

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| OnCOB\_ID\_SyncUpdate | 输入 | CO\_Data指针 | | Canopen设备地址 |
| indextable指针 | |  |
| UNS8变量 | |  |
| 输出 | UNS32变量 | |  |
| 使用 |  | | |
| startSYNC | 输入 | CO\_Data指针 | Canopen设备地址 | |
| 使用 | 本设备启动SYNC发送  调用RegisterSetODentryCallBack()函数 | | |
| 调用SetAlarm()函数 | | |
| stopSYNC | 输入 | CO\_Data指针 | Canopen设备地址 | |
| 使用 | 本设备停止SYNC发送  调用RegisterSetODentryCallBack()函数 | | |
| 调用DelAlarm()函数 | | |

# PDO

## PDO服务器

|  |  |
| --- | --- |
| 一个PDO服务器由1个TPDO子服务器和1个RPDO子服务器组成  PDO服务器的序号从**0**开始计起  一个PDO服务器包含五个部分：  (1)TPDO通讯参数；  (2)TDPO映射参数；  (3)RPDO通讯参数；  (4)RDPO映射参数；  (5)一个PDO服务器对应一个s\_PDO\_status变量 | 如**0号PDO服务器**对应：  (1)TPDO1通讯参数对象🡪ObjDict\_objdict[ObjDict\_firstIndex. PDO\_TRS]  (2)TPDO1 映射参数对象🡪ObjDict\_objdict[ObjDict\_firstIndex. PDO\_TRS\_MAP]  (3)RPDO1通讯参数对象🡪ObjDict\_objdict[ObjDict\_firstIndex.PDO\_RCV\_MAP]  (4)RPDO1映射参数对象🡪ObjDict\_objdict[ObjDict\_firstIndex.PDO\_RCV\_MAP]  (5)PDO\_status[0] |
| 如**1号PDO服务器**对应：  (1)TPDO2通讯参数对象🡪ObjDict\_objdict[ObjDict\_firstIndex. PDO\_TRS+1]  (2)TPDO2 映射参数对象🡪ObjDict\_objdict[ObjDict\_firstIndex. PDO\_TRS\_MAP+1]  (3)RPDO2通讯参数对象🡪ObjDict\_objdict[ObjDict\_firstIndex.PDO\_RCV\_MAP+1]  (4)RPDO2映射参数对象🡪ObjDict\_objdict[ObjDict\_firstIndex.PDO\_RCV\_MAP+1]  (5)PDO\_status[1] |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| s\_PDO\_status  指针 | 成员 | 类型 |  | |
| transmit\_type\_parameter | UNS8  变量 | bit | 说明 |
|  |  |
|  |  |
|  |  |
|  |  |
| 0 |  |
| event\_timer | TIMER\_HANDLE  变量 | 该PDO服务器的事件计时任务的状态，初始值=TIMER\_NONE | |
| inhibit\_timer | TIMER\_HANDLE  变量 | 该PDO服务器的禁止时间任务的状态，初始值=TIMER\_NONE | |
| last\_message | Message  变量 | 记录该PDO服务器上一次发送的PDO报文  初始值 = {0,0,0,{0,0,0,0,0,0,0,0}} | |

|  |  |  |
| --- | --- | --- |
| CO\_Data: | | |
| PDO\_status | s\_PDO\_status  指针 | PDO服务器组  创建s\_PDO\_status类型数组，传递数组的首地址  数组的每一个元素表示一个PDO服务器；  初始值都设为 |
| RxPDO\_EventTimers | TIMER\_HANDLE  指针 |  |
| RxPDO\_EventTimers\_Handler | void指针 |  |
| post\_TPDO | post\_TPDO\_t  变量 | typedef void (\*post\_TPDO\_t)(CO\_Data\*);  创建上述类型的函数，传递函数地址，即函数名 |

## 发送PDO报文

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| CopyBits | 输入 | UNS8变量 | 要copy的数据的bit数目total\_bits | |
| UNS8指针 | copy源数组：输入一个UNS8类型的数组的某个元素的地址  该数组的内容是复制的源数据，该元素是第一个要复制的数据，  (从低位开始向高位提供复制的数据) | |
| UNS8变量 | 上一个变量已经确定了要复制哪一个变量  本参数表示接下来要复制该变量的第几位(从低位开始向高位复制)  取值：0~7 | |
| UNS8变量 | 0：little endian 1：big endian | |
| UNS8指针 | copy目标数组：输入一个UNS8类型的数组的某个元素的地址  该数组用于存储复制得到的数据，该元素是第一个用于存储数据的元素  (从低位开始向高位接收复制的数据) | |
| UNS8变量 | 上一个变量已经确定了要把复制得到的数据存到哪一个变量  本参数表示接下来用于存储的是该变量的第几位(从低位开始向高位接收)  取值：0~7 | |
| UNS8变量 | 0：little endian 1：big endian | |
| 使用 | 本函数用于复制数据(数组复制)，  一次只能复制copy源数组的某**一个**元素 到 copy目标数组的某**一个**元素；  显然一次最多只能复制8bit数据；  如果NbBits＞8，那么就需要分多次复制共total\_bits位的数据，使用一个while循环  在其中的**某一次复制中**： | | |
| 复制的位数BitsToCopy，由剩余位数小的那一方决定，如某一次复制：  copy源数组从某一元素的第3位开始；表明本次复制最多复制5bit  copy目标数组从某一元素的第6位开始；表明本次复制最多复制2bit； 综上，BitsToCopy = 2 | | |
| 根据本次复制：copy源数据的开始位(如3)，以及复制的bit数目BitsToCopy(如2)  就可以确定本次复制该copy源数据的哪些位(bit3 ~~bit4)； | | |
| 一次复制结束后，copy源数组和copy目标数组都各自取下一个元素，开始下一次复制； | | |
| buildPDO | 输入 | CO\_Data指针 | | Canopen设备地址 |
| UNS8变量 | | PDO服务器序号(从0开始) |
| Message指针 | | 创建一个空白的Message变量，用于存储要发送的PDO报文  输入该Message变量的地址； |
| 使用 | CAN-ID | | 根据输入的PDO服务器序号获取要使用的 TPDO通讯参数对象  获取01条目的值，低11位就是该PDO报文的CAN-ID |
| RTR | | rtr=0  是CAN数据帧报文 |
| 数据场 | | 根据输入的PDO服务器序号获取要使用的 TPDO映射参数对象  遍历该映射参数对象的所有条目(除了00条目)，每一个条目都包含发送数据：  **所在对象的索引、所在条目的子索引**、**位宽(数据定义)**  调用getODentry()函数获取该条目的对应的内容  调用CopyBits()函数，将getODentry()获取的内容复制到输入的空白Message变量中 |
| sendPdo | 输入 | CO\_Data指针 | | Canopen设备地址 |
| UNS32变量 | | PDO服务器序号(从0开始) |
| Message指针 | | 输入一个Message变量的地址，该变量是要发送的PDO报文 |
| 使用 | (1)**设置该PDO服务器的last\_message为输入的Message变量**  (2)调用canSend()函数发送该PDO报文 | | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| sendOnePDOevent | | 输入 | | CO\_Data指针 | | Canopen设备地址 |
| UNS8变量 | | PDO服务器序号(从0开始) |
| 查看  通讯状态 | | 首先查看设备的当前通讯状态，是否启动了PDO传输  没有启动🡪函数返回； 启动🡪继续执行下面的代码 | | |
| 查看  禁止标志 | | 查看该PDO服务器的transmit\_type\_parameter，是否含有禁止标志  含有禁止标志🡪函数返回； 没有禁止标志🡪函数可以继续执行 | | |
| 创建  PDO报文 | | 调用buildPDO()函数创建的PDO报文，  将该PDO报文与上一次发送的PDO报文(\*->last\_message)比较  两者相等🡪函数返回； 两者不相等🡪函数继续执行 | | |
| 事件计时 | | 根据输入的PDO服务器序号获取要使用的**TPDO通讯参数对象的05条目**，  获取事件计时器的数值EventTimerDuration，查看事件计时功能是否启用；  启用(EventTimerDuration≠0)🡪调用SetAlarm()函数，事件计时任务开始新一轮的计时 | | |
| 禁止时间 | | 根据输入的PDO服务器序号获取要使用的**TPDO通讯参数对象的03条目**，  获取禁止时间的数值InhibitTimerDuration查看禁止时间功能是否启用，  启用(InhibitTimerDuration≠0)🡪调用SetAlarm()函数，禁止时间任务开始新一轮的计时  配置该PDO服务器的transmit\_type\_parameter，**加入禁止标志**  事件计时任务和禁止时间任务 定时的起始时间是总是相同的； | | |
| 发送PDO报文 | | 调用sendPDO()函数发送新创建的PDO报文 | | |
| PDOEventTimerAlarm | | 当启用事件计时功能时(EventTimerDuration≠0)，本函数才有效，  每当事件定时任务溢出，就会执行**PDOEventTimerAlarm()**函数，该函数的作用是：  (1)初始化该PDO服务器的\*->last\_message .cob\_id  (2)调用**sendOnePDOevent()**函数发送PDO报文： | | | | |
| 查看  通讯状态 | | 首先查看设备的当前通讯状态，是否启动了PDO传输  如果没有启动，直接返回0，如果启动了，继续执行下面的代码 | | |
| 查看  禁止标志 | | 查看该PDO服务器的transmit\_type\_parameter，是否含有禁止标志  如果含有禁止标志，函数返回；如果没有禁止标志，函数可以继续执行 | | |
| 创建  PDO报文 | | 调用buildPDO()函数创建的PDO报文，  将该PDO报文与上一次发送的PDO报文(\*->last\_message)比较  由于本函数函数已经将该PDO服务器的last\_message .cob\_id清0，  所以新创建的PDO报文的cob\_id肯定不为0，所以两者肯定不相等，函数继续执行； | | |
| 事件计时 | | 调用SetAlarm()函数，事件计时任务开始新一轮的计时 | | |
| 禁止时间 | | 查看禁止时间功能是否启用，如果启用(InhibitTimerDuration≠0)，那么：  调用SetAlarm()函数，禁止时间任务开始新一轮的计时  配置该PDO服务器的transmit\_type\_parameter，**加入禁止标志** | | |
| 发送PDO报文 | | 调用sendPDO()函数发送新创建的PDO报文 | | |
| 事件计时功能：每经过指定的时间**周期发送PDO报文**  **注意**sendOnePDOevent()函数严格规定了该PDO服务器发送PDO报文的间隔，也就是说，  如果不是在事件计时器所规定的间隔点发送该PDO报文  (即某次执行sendOnePDOevent()函数不是由PDOEventTimerAlarm()调用的)  那么执行sendOnePDOevent()函数时，创建的PDO报文与上一次发送的PDO报文就是相同的，sendOnePDOevent()返回0，未能成功发送PDO | | | | |
| PDOInhibitTimerAlarm | | 当启用禁止时间功能时(InhibitTimerDuration≠0)，本函数才有效，  每当禁止时间任务溢出，就会执行**PDOInhibitTimerAlarm()**函数，该函数的作用是：  (1)初始化该PDO服务器的last\_message  (2)配置该PDO服务器的transmit\_type\_parameter，**清除禁止标志**；  (3)再次调用**sendOnePDOevent()**函数发送PDO报文 | | | | |
| 查看  通讯状态 | | 首先查看设备的当前通讯状态，是否启动了PDO传输  如果没有启动，直接返回0，如果启动了，继续执行下面的代码 | | |
| 查看  禁止标志 | | 查看该PDO服务器的transmit\_type\_parameter，是否含有禁止标志  由于禁止标志已经被清除，所以函数可以继续执行 | | |
| 创建  PDO报文 | | 调用buildPDO()函数创建的PDO报文  将该PDO报文与上一次发送的PDO报文(\*->last\_message)比较，  如果不相等，函数可以继续执行 | | |
| 事件计时 | | 查看事件计时功能是否启用，如果启用(EventTimerDuration≠0)  那么调用SetAlarm()函数，事件计时任务开始新一轮的计时 | | |
| 禁止时间 | | 调用SetAlarm()函数，禁止时间任务开始新一轮的计时，  配置该PDO服务器的transmit\_type\_parameter，**加入禁止标志** | | |
| 发送  PDO报文 | | 调用sendPDO()函数发送新创建的PDO报文 | | |
| 禁止时间功能：两个相同的PDO报文发送时间间隔不能小于禁止时间  如果某个PDO服务器发送一条PDO报文后，马上又调用sendOnePDOevent发送TPDO报文  此时该PDO服务器是处于禁止发送状态，sendOnePDOevent()返回0，未能成功发送PDO | | | | |
| \_sendPDOevent | | 输入 | | CO\_Data指针 | | Canopen设备地址 | |
| UNS8变量 | | **1，表示同步传输** | |
| 获取  通讯类型 | | 使用while循环遍历所有的PDO服务器，查看它们的**TPDO通讯参数对象的条目2**，  获取通讯类型，针对不同的通讯类型执行相应的操作： | | | |
| 1~240 | | 表明该PDO服务器的传输类型是：**同步循环**  先更新该PDO服务器的同步接收值：++transmit\_type\_parameter  再获取该PDO服务器的同步溢出值：即TPDO通讯参数对象条目2的数值 | | | |
| 如果同步接收值达到了设定的同步溢出值，那么：  初始化该PDO服务器的同步接收值，transmit\_type\_parameter=0  调用buildPDO()函数创建PDO报文()  调用sendPDO()函数发送该PDO报文  进入下一个PDO服务器 | | | |
| 如果同步接收值＜同步溢出值，那么：不发送PDO报文  进入下一个PDO服务器 | | | |
| 0 | | 表明该PDO服务器的传输类型是：**同步非循环**  即该设备每收到一个同步报文，该PDO服务器就会发送PDO报文  调用sendOnePDOevent()函数发送PDO报文  进入下一个PDO服务器 | | | |
| 252 | | 表明该PDO服务器的传输类型是：**同步循环远程帧**  调用buildPDO()函数创建PDO报文，**仅创建但不发送！**  如果创建成功，配置transmit\_type\_parameter，加入同步就绪标志(最低位=1) | | | |

## 接收PDO报文

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| proceedPDO | 输入 | CO\_Data指针 | | Canopen设备地址 |
| Message指针 | | 创建一个Message变量，存储从机收到的来自主机的CAN报文  输入该Message变量的地址； |
| 输出 | UNS8变量 | |  |
| 本函数在canDispatch()函数中被调用，当设备收到CAN报文，就会进入CAN中断，  然后执行CAN接收中断服务函数中的canDispatch()函数，判断所收到的CAN报文的功能码，如果功能码是：  PDO1tx(#define PDO1tx 0x3) PDO1rx(#define PDO1rx 0x4) PDO2tx(#define PDO2tx 0x5)  PDO2rx(#define PDO2rx 0x6) PDO3tx(#define PDO3tx 0x7) PDO3rx(#define PDO3rx 0x8)  PDO4tx(#define PDO4tx 0x9) PDO4rx(#define PDO4rx 0xA)  那么就会执行本函数： | | | |
| 查看收到的CAN报文的RTR，如果该报文是**远程帧(rtr=0)**，那么：  表明本设备是作为PDO的消费者，收到了来自其他设备的PDO报文 | | | |
| 查找  PDO服务器 | | 使用while循环遍历所有的PDO服务器，查看它们的**RPDO通讯对象**的01条目，获取基础CAN-ID  直到找到一个CAN-ID与收到的PDO报文的CAN-ID相同的**RPDO通讯对象**  那么就使用该PDO服务器来接收PDO报文 | |
| 获取  事件计时器  数值 | | 根据输入的PDO服务器序号获取要使用的**RPDO通讯参数对象的05条目**，  获取其事件计时器的数值EventTimerDuration | |
| 接收PDO | | 根据该PDO服务器的 RPDO映射参数对象，从该对象的01条目开始，一直到最后一个条目：  高24位：将用来存储该PDO报文的对象的索引和条目的子索引  低8位：可以存储的位宽  调用CopyBits()函数，获取该PDO报文的内容，到一个临时缓冲区temp[]  调用setODentry()函数，将临时缓冲区temp[]传入到指定的对象条目中 | |

## 同步触发

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| proceedSYNC | 本函数在canDispatch()函数中被调用，当设备收到CAN报文，就会进入CAN中断，然后执行CAN接收中断服务函数中的canDispatch()函数；canDispatch()函数中，判断所收到的CAN报文的功能码，  如果功能码是SYNC(#define SYNC 0x1 ，EMCY和SYNC共用) ，并且：  查看设备此刻的通讯状态，SYNC处于启动状态：\*->CurrentCommunicationState.csSYNC=1  那么就会执行本函数； | | | |
| 设备接收同步报文，用于触发PDO传输，查看设备此刻的通讯状态： 如果PDO处于禁止状态：\*->CurrentCommunicationState.csPDO=0，  那么直接返回0  如果PDO处于启动状态，那么就会调用**\_sendPDOevent()**函数 和 **\*->post\_TPDO()**函数  \_sendPDOevent()函数用于发送PDO报文：输入的第二个参数是1，表明是同步传输； | | | |
| 调用\*->post\_sync()函数 | | | |
| sendSYNC | 输入 | CO\_Data指针 | Canopen设备地址 |
| 输出 | UNS8变量 |  |
| 使用 | 先调用sendSYNCMessage()函数，  再调用proceedSYNC()函数 | |

## 远程帧触发

|  |  |  |  |
| --- | --- | --- | --- |
| sendPDOrequest | 输入 | CO\_Data指针 | Canopen设备地址 |
| UNS16变量 | 输入要使用的RPDO通讯对象的16位索引号 |
| 使用 | 本设备作为PDO消费者，接收其他设备的PDO报文  输入的16位索引号是本设备要使用的RPDO通讯对象的索引号；  该RPDO通讯对象的01条目，取低16位：  是该PDO远程帧的CAN-ID  是对应的PDO生产者的某个PDO服务器的TPDO通讯参数01条目存储的COB-ID；  调用canSend()函数发送该PDO远程帧； | |
|  | | | |
| proceedPDO | 输入 | CO\_Data指针 | Canopen设备地址 |
| Message指针 | 创建一个Message变量，存储从机收到的来自主机的CAN报文  输入该Message变量的地址； |
| 输出 | UNS8变量 |  |
| 本函数在canDispatch()函数中被调用，当设备收到CAN报文，就会进入CAN中断，  然后执行CAN接收中断服务函数中的canDispatch()函数，判断所收到的CAN报文的功能码，如果功能码是：  PDO1tx(#define PDO1tx 0x3) PDO1rx(#define PDO1rx 0x4) PDO2tx(#define PDO2tx 0x5)  PDO2rx(#define PDO2rx 0x6) PDO3tx(#define PDO3tx 0x7) PDO3rx(#define PDO3rx 0x8)  PDO4tx(#define PDO4tx 0x9) PDO4rx(#define PDO4rx 0xA)  那么就会执行本函数： | | |
| 查看收到的CAN报文的RTR，如果该报文是**远程帧(rtr=1)**，那么：  表明本设备是作为PDO的生产者，要进行 **远程帧** 类型的PDO传输 | | |
| 查找  PDO服务器 | 使用while循环遍历所有的PDO服务器，查看它们的**TPDO通讯对象**的01条目，获取基础CAN-ID  直到找到一个CAN-ID与收到的PDO远程帧的CAN-ID相同的**TPDO通讯对象**  那么就使用该PDO服务器来发送PDO报文 | |
| 同步  远程帧 | 查看该PDO服务器的通讯类型，如果是252🡪同步远程帧，那么  查看该PDO服务器是否处于同步就绪状态  是则调用canSend()函数，发送PDO报文，该PDO报文就是该PDO服务器的last\_message，  它是该PDO服务器收到同步报文时，调用\_sendPDOevent()函数创建好了的 | |
| 事件  远程帧 | 查看该PDO服务器的通讯类型，如果是253🡪事件远程帧，那么  调用buildPDO()函数创建PDO报文()  调用canSend()函数发送该PDO报文 | |
| 事件 | 查看该PDO服务器的通讯类型，如果是254🡪制造商事件，或255🡪CIA协议事件，那么  删除该PDO服务器的事件计时任务  删除该PDO服务器的禁止时间任务  调用**PDOEventTimerAlarm()函数** | |

## 启动传输

|  |  |  |  |
| --- | --- | --- | --- |
| PDOInit | 输入 | CO\_Data指针 | Canopen设备地址 |
| 配置  回调函数 | 使用while循环遍历所有PDO服务器 | |
| 初始化  PDO传输 | 调用\_sendPDOevent()函数，第二个参数输入**0**，表示事件触发 | |
|  | | | |
| PDOStop | 输入 | CO\_Data  指针 | Canopen设备地址 |
| 使用 |  | |

|  |  |  |  |
| --- | --- | --- | --- |
| TPDO\_Communication\_Parameter\_Callback | 输入 | CO\_Data指针 | Canopen设备地址 |
| indextable指针 |  |
| UNS8变量 |  |
|  |  | |
|  |  | |

|  |  |  |  |
| --- | --- | --- | --- |
| \_sendPDOevent | 输入 | CO\_Data指针 | Canopen设备地址 |
| UNS8变量 | **输入0，表示事件触发** |
| 获取  通讯类型 | 使用while循环遍历所有的PDO服务器，查看它们的**TPDO通讯参数对象的条目2**，  获取通讯类型，针对不同的通讯类型执行相应的操作： | |
| 事件触发 | 254或255 | 表明该PDO服务器的传输类型是：制造商事件或CIA协议事件  查看该PDO服务器的transmit\_type\_parameter，是否含有禁止标志  因为之前从未使用过sendOnePDOevent()函数来发送PDO报文，  所以 \*->transmit\_type\_parameter 为初始值0，没有禁止标志；  函数继续执行：调用sendOnePDOevent()函数发送PDO报文  进入下一个PDO服务器 |
| 其他 | 直接进入下一个PDO服务器 |
| sendOnePDOevent | 查看  通讯状态 | 首先查看设备的当前通讯状态，是否启动了PDO传输  如果没有启动，直接返回0，如果启动了，继续执行下面的代码 | |
| 查看  禁止标志 | 查看该PDO服务器的transmit\_type\_parameter，是否含有禁止标志  因为之前从未使用过sendOnePDOevent()函数来发送PDO报文，  所以 \*->transmit\_type\_parameter 为初始值0，没有禁止标志，函数继续执行 | |
| 创建  PDO报文 | 调用buildPDO()函数创建的PDO报文，  将该PDO报文与上一次发送的PDO报文(\*->last\_message)比较，  因为之前从未使用过sendOnePDOevent()函数来发送PDO报文，  所以 \*->last\_message .cob\_id 为初始值0，新创建的PDO报文的cob\_id肯定不为0，  所以两者肯定不相等，函数继续执行 | |
| 事件计时 | 查看事件计时功能是否启用，如果启用(EventTimerDuration≠0)  那么调用SetAlarm()函数创建事件计时任务，事件计时任务开始第一轮的计时  定时时间长度是EventTimerDuration，回调函数是PDOEventTimerAlarm； | |
| 禁止时间 | 查看禁止时间功能是否启用，如果启用(InhibitTimerDuration≠0)  那么调用SetAlarm()函数创建禁止时间任务，禁止时间任务开始第一轮的计时  定时时间长度是InhibitTimerDuration，回调函数是PDOInhibitTimerAlarm；  配置该PDO服务器的transmit\_type\_parameter，**加入禁止标志** | |
| 发送  PDO报文 | 调用sendPDO()函数发送新创建的PDO报文  即第一个PDO报文 | |

# LSS

## 参数配置

|  |  |  |
| --- | --- | --- |
| LSS参数预定义  位置：def.h | | |
| 名称 | Value | 说明 |
| LSS\_RESET |  |  |
| LSS\_FINISHED |  |  |
| LSS\_ABORTED\_INTERNAL |  |  |
| LSS\_TRANS\_IN\_PROGRESS |  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| CO\_Data | 成员 | 函数类型 | | |  |
| lss\_StoreConfiguration | lss\_StoreConfiguration\_t | 输入 | CO\_Data指针 |  |
| UNS8指针 |
| UNS8指针 |
| lss\_transfer | lss\_transfer\_t变量 |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| CO\_Data | lss\_transfer | lss\_transfer\_t变量 |  |
| lss\_StoreConfiguration | lss\_StoreConfiguration\_t  变量 |  |