PMBus™协议栈用户手册

版本1.0

内容

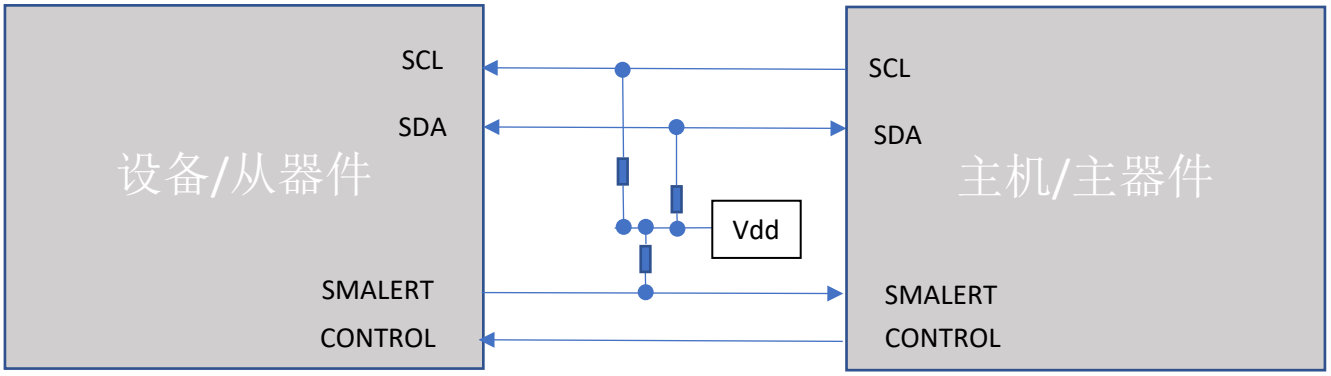
1.0 概述

从器件/设备的PMBus协议栈是使用dsPIC®“C”系列器件的片上I2C外设实现的。它表示一个代码块，用于实现协议的网络链路层。

2.0 硬件连接

PMBus主机和设备之间的硬件连接。

CONTROL信号是可选的，未与“从器件/设备”协议栈集成。应用可以根据需要实现所需的功能。



2.1 使用的单片机资源/外设

- I2C + DMA → I2C模块配置为从模式，7位寻址模式。
  - 默认情况下使用START、STOP、地址保持（AHEN）和数据保持（DHEN）。默认情况下，SMBus IO电压阈值选择处于禁止状态。
  - CRC → 计算传入数据包或传出数据包的数据包错误校验（Packet Error Checking，PEC）。
  - 选择内部快速RC（Fast RC，FRC）作为主振荡器。
  - UART → 将调试消息转储到终端上。波特率配置为**默认值230400**。

3.0 协议栈概述

PMBus从协议栈可识别并处理以下SMBus协议：

- SEND_BYTE
- RECEIVE_BYTE
- WRITE_BYTE
- READ_BYTE
- WRITE_WORD
- READ_WORD
- WRITE_BLOCK
- READ_BLOCK
- PROCESS CALL
- BLOCK WRITE
- BLOCK READ
- BLOCK READ/WRITE PROCESS CALL
- 组命令

PEC字节是**可选的**，可根据应用需求进行配置。

注：有关上述协议的更多信息，请参见SMBus规范2.0

3.1 SMBus协议——如何通过协议栈进行处理

主应用程序以非常简单的方式接收和发送信息。

从主机接收到WRITE命令后，PMBus协议栈将调用应用程序处理程序（**PMBus_Device_Appl_CallBack**）复制接收到的有效负载。之后应用程序可以执行必要的操作。

任何READ命令都是自动执行的。也就是说，从主机接收到READ命令后，从协议栈将读取与该命令关联的变量，并将值发送给主机。

对于PROCESS CALL，将在接收到“WRITE”操作的有效负载后调用用户应用程序。

4.0 设备故障管理

PMBus协议为用户提供了丰富的工具集，用于监视操作以及管理PMBus设备中的故障或警告。主机可以使用READ命令向PMBus设备询问其当前状态。此外，每个参数（例如输出电压、设备温度或输出电流）都对应一个READ命令。PMBus协议还具有为电源转换器件的每个重要方面编程故障或警告级别的能力。

PMBus协议支持两种警报设置：

A. 警告阈值作为次要警报
   警告条件用于指示设备侧发生问题，但不影响设备的正常运行。

B. 故障阈值作为主要警报
   故障是比警告更严重的事件，可能导致设备禁止输出级并停止将能量传递到输出。

   值得一提的一个重要方面是，PMBus协议提供相应的命令来设置对每种故障条件的响应。这些命令包含一个数据字节，用于指示设备如何响应故障。这些故障响应命令均要求用户从设备响应故障条件的三种方式中选择一种。

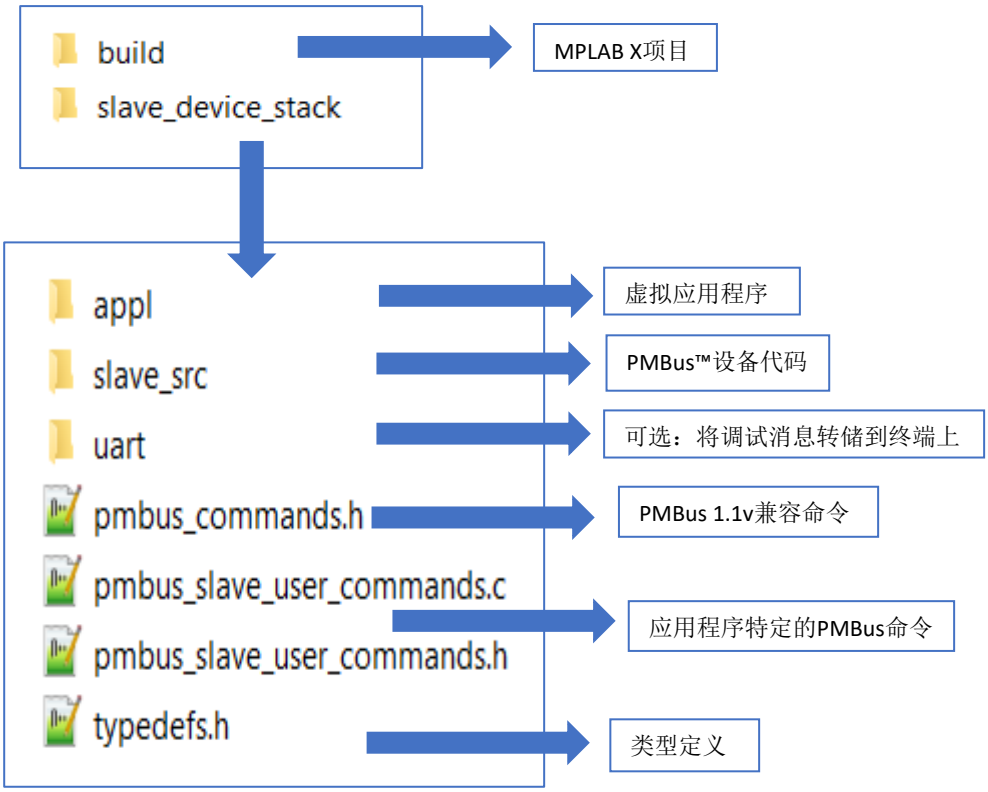
在发生警告或故障事件之后，PMBus设备可通过两种方式向主机通知其状态：

1. 将STATUS寄存器中的警告/故障条件位置1，并等待主机轮询它们
2. 通过以下方法之一通知主机已发生警告/故障情况：
   - SMBAlert线
     这是一条可选的中断线，从器件可以选择使用该中断线来向主器件通知警告或故障情况。
   - 主机通知协议
     使用该SMBUS协议，从器件可以临时控制总线，并充当总线主器件与主机通信。

该协议栈仅支持SMBAlert线。不支持“主机通知协议”。

5.0 软件协议栈概述

5.1 文件夹结构：

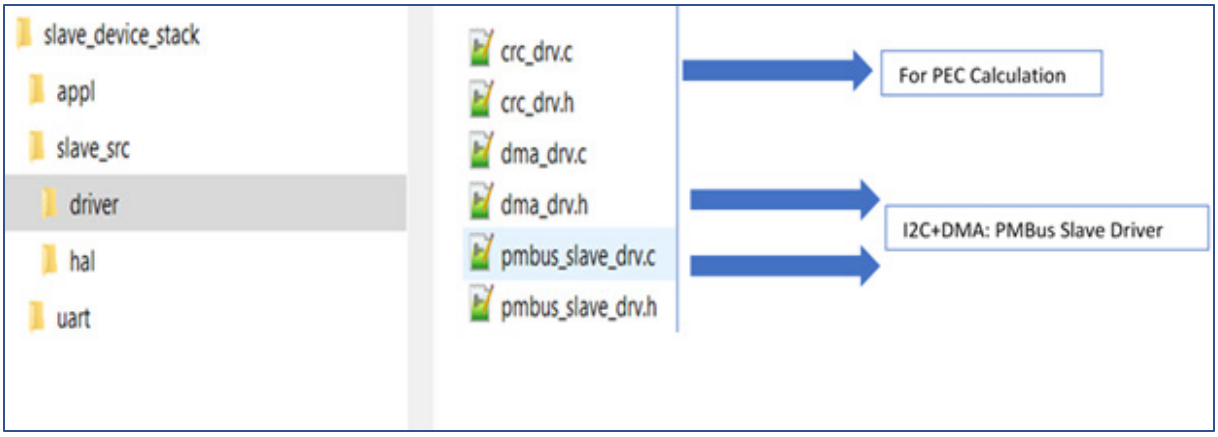


5.1.1 slave_src文件夹

5.1.1.1 驱动程序文件夹

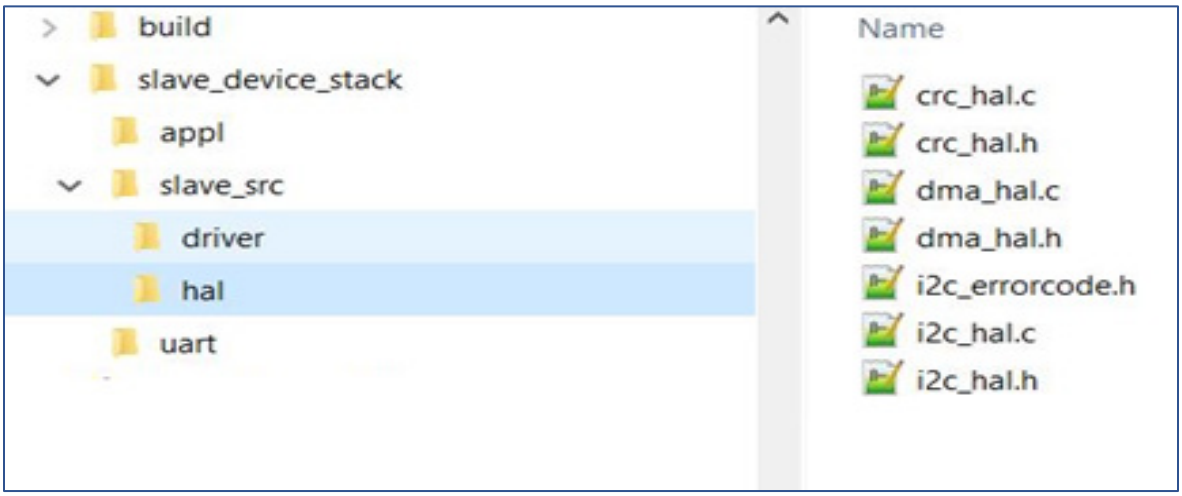
驱动程序文件夹包含以下外设的设备驱动程序

- CRC：计算数据包错误校验（PEC）。
- I2C + DMA：pmbus_slave_driver.c/.h使用I2C外设并实现符合1.1的协议栈。

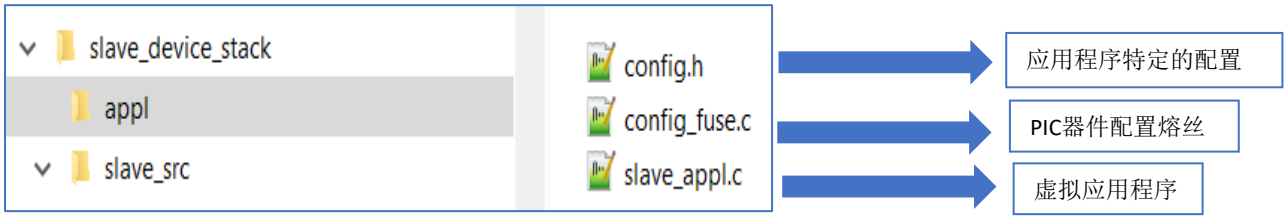


5.1.1.2 hal文件夹

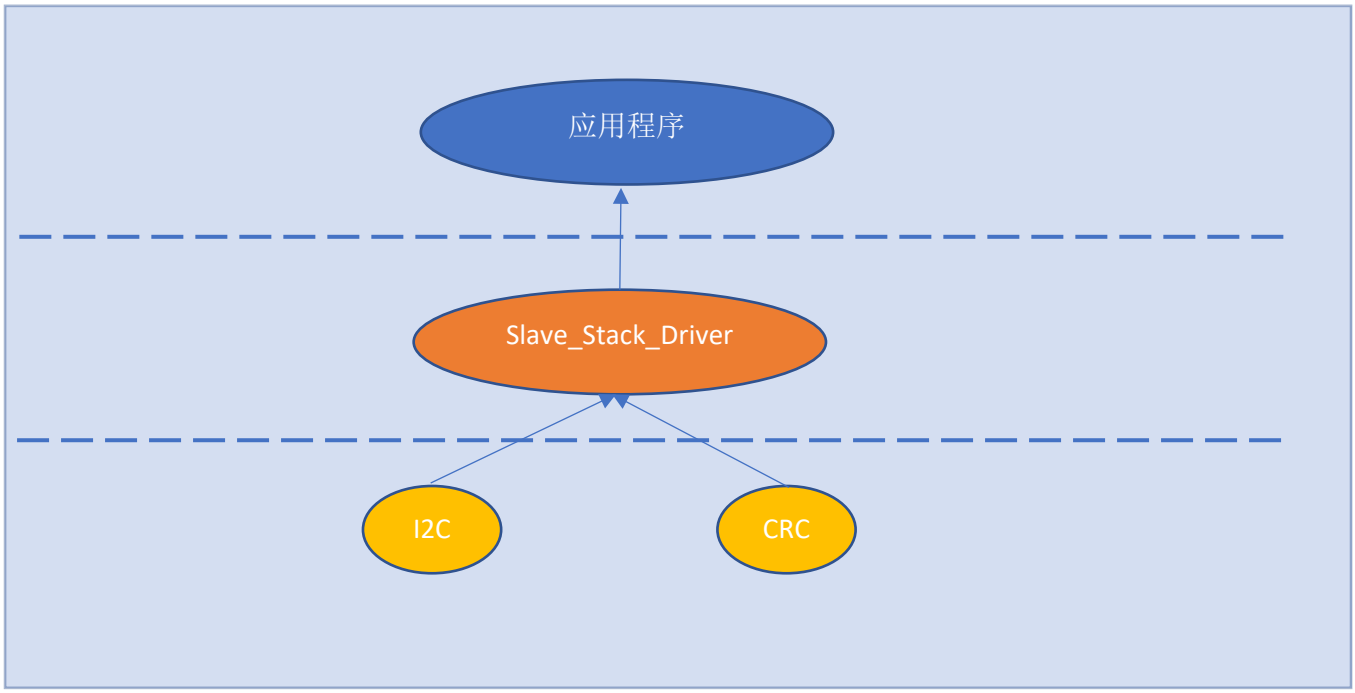
hal是片上外设的硬件抽象层（**H**ardware **A**bstraction **L**ayer，HAL）。



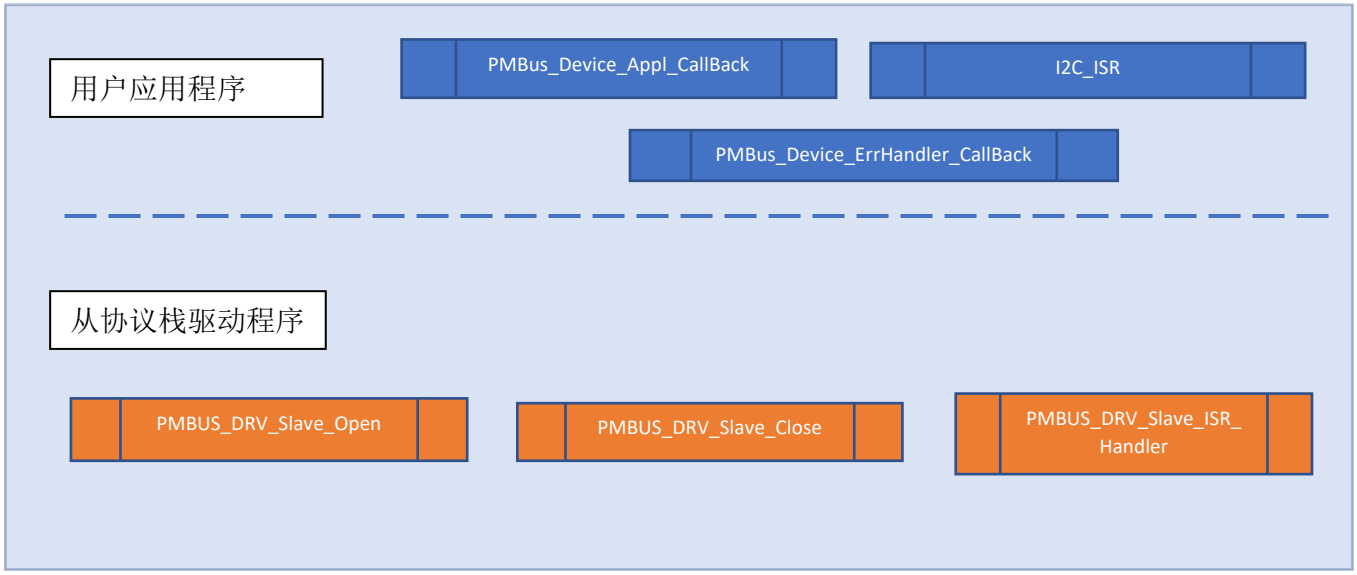
5.1.2 应用程序文件夹



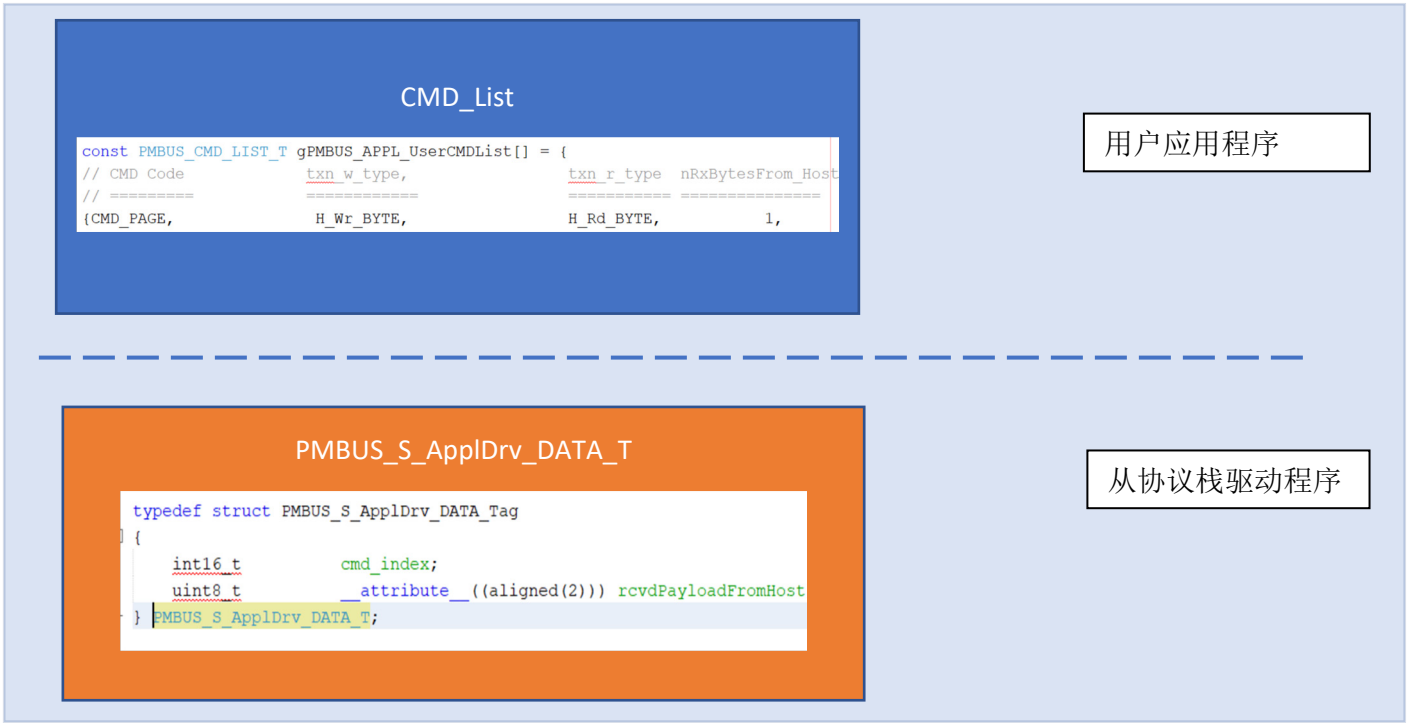
6.0 层间交互



6.1 函数接口



6.2 数据接口

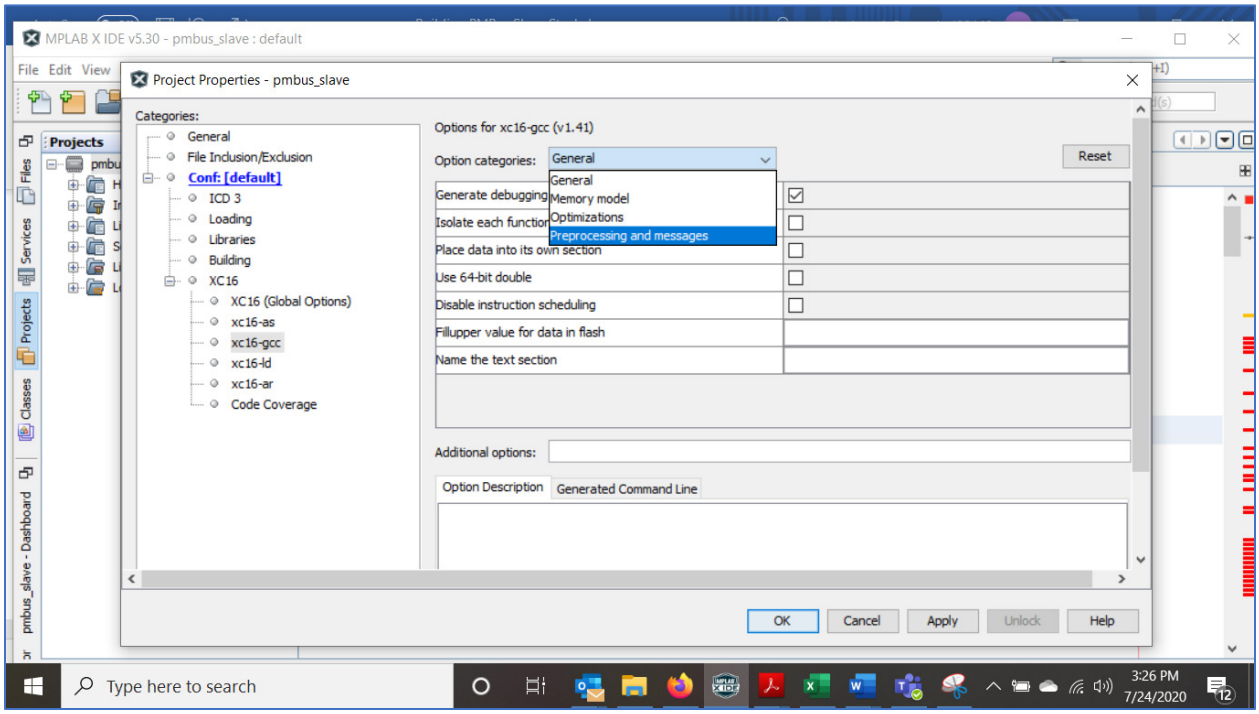


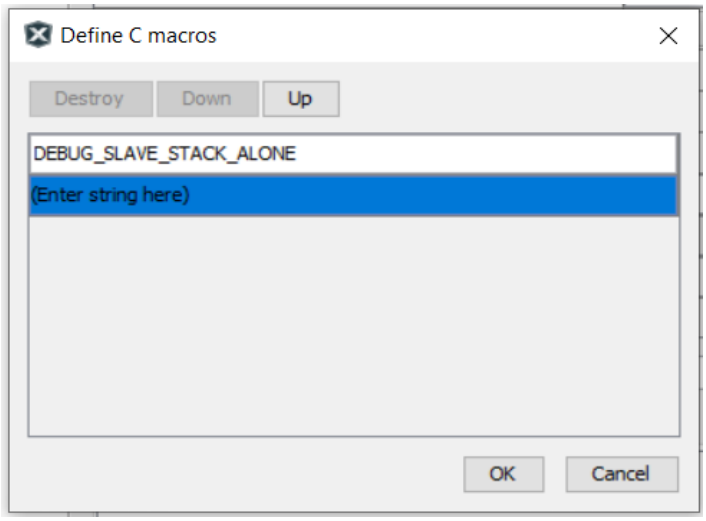
7.0 配置协议栈

7.1 MPLAB X项目编译中的配置

7.1.1 编译选项

xc16-gcc → Preprocessing and messages → Define C macros（xc16-gcc → 预处理和消息 → 定义C宏）

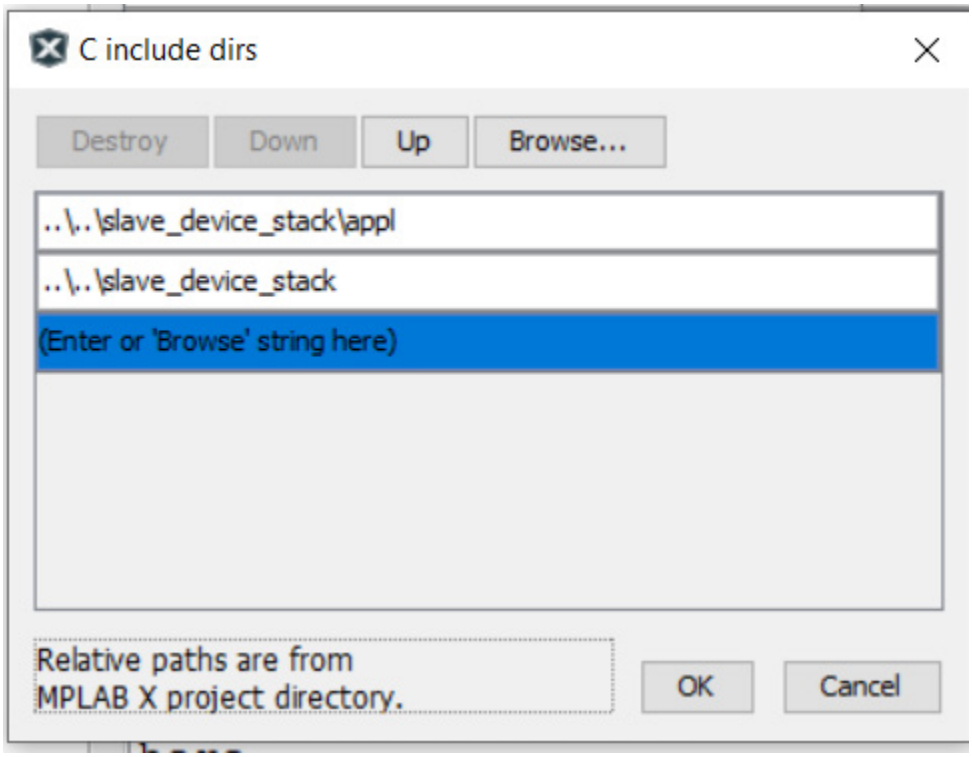




7.1.2 包含目录

包含目录用于应用程序特定的“config.h”和“typedefs.h”。

xc16-gcc → Preprocessing and messages → C Include dirs（xc16-gcc → 预处理和消息 → C包含目录）



7.2 config.h中的配置

应用程序特定的配置在appl/config.h中完成。

7.2.1 外设实例配置

```
/*******************************************************************************
 Peripheral Instances Configuration
 * ---------------------------------------------------------------------------
-----------------------------------------------------------------------------
 * Define - Debug Print Option
 ******************************************************************************/

    #define mS_I2C_INSTANCE           2       // I2C2 is used.
    #define mDMA_CH_INSTANCE_FOR_I2C  0       // Channel 0 is used.
    #define mUART_INSTANCE            1       // UART 1 module is used.

    #define mPRINT_DEBUG_MESSAGE              // To print debug messages on Terminal
```

7.2.2 PMBus从地址和PEC配置

```
/*****************************************************************
 PMBus I2C Configuration
 *****************************************************************
    #define mPMBus_SLAVEDEVICE_ADDR     (0x3C)
    #define mPEC_USED  // If PEC is used by User application
```

7.2.3 SMALERT IO配置

```
/*****************************************************************
SMALERT IO Configuration.
*****************************************************************

#define mPMBUS_S_SMALERT_LAT (LATE)
#define mPMBUS_S_SMALERT_LAT_BitPostn        (6)
#define mPMBUS_S_CONFIG_SMALERT_TRIS(x)      (_TRISE6 = x)
#define mPMBUS_S_CONFIG_SMALERT_LAT(x)       (_LATE6 = x)
#define mPMBUS_S_CONFIG_SMALERT_ODC(x)       (_ODCE6 = x)
```

7.2.4 配置DMA边界地址

有关DMA边界地址的信息，请参见器件数据手册。

```
/*****************************************************************
 DMA Upper and Lower Boundary and Channel Configuration.
 *****************************************************************
#define mDMA_UPPER_RAM_ADDRESS    (0x5000)
#define mDMA_LOWER_RAM_ADDRESS    (0x550)
```

7.2.5 I2C的DMA触发源

有关触发源的信息，请参见器件数据手册。根据I2C实例选择，需要选择DMA触发源。

```
#if (mS_I2C_INSTANCE == 1)
    #define mDMA_Trigger_FROM_I2C    (0x09)
#elif (mS_I2C_INSTANCE == 2)
    #define mDMA_Trigger_FROM_I2C    (0x13)
#elif (mS_I2C_INSTANCE == 3)
    #define mDMA_Trigger_FROM_I2C    (0x64)
#else
    #error "Undefined I2C instance in config.h"
#endif
```

7.2.6 UART配置

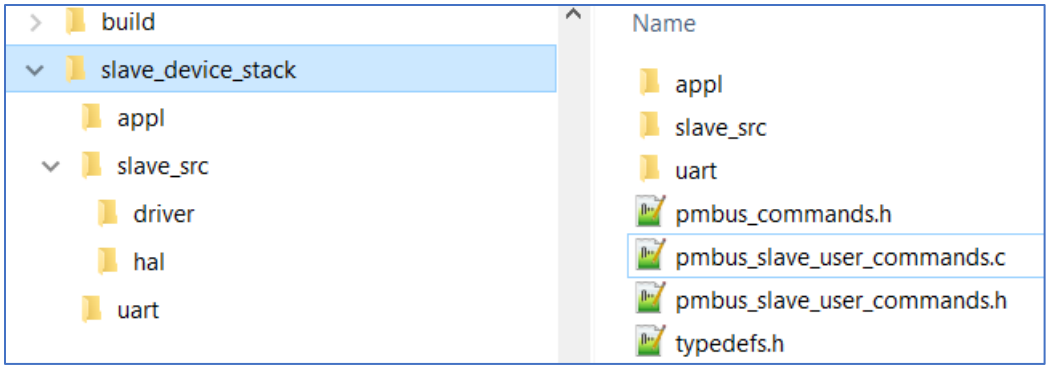
- UART缓冲区大小、波特率和UARTx Tx引脚配置。
- 使用输出重映射值重映射UARTx Tx引脚。有关PPS引脚和输出重映射值，请参见器件数据手册。在以下示例中，RP70映射到UART1Tx。从器件数据手册中可知，UART1Tx输出重映射值为1。

```
/*****************************************************
 UART Module.
 *****************************************************

#define mUART_BUFF_SIZE                 (1024)
#define mUART_BAUD                      (230400)
#define mSetUART_Tx_TRISAsOutput()      (_TRISD6 = 0)
#define mUART_RPOR                      (_RP70R)
#define mUART_Output_Remap_Value        (1) // For UART1
```

8.0 应用程序中的PMBus“命令”配置

可以在pmbus_slave_user_commands.c/.h文件中定义由客户应用程序实现的命令。这些文件位于“root”文件夹下。



8.1 PMBus规范1.1命令

所有命令都在**pmbus_command.h**中定义。

```
/*****************************************************
 * PMBus 1.1 compatible Commands and Bus Protocols.
  *****************************************************/
typedef enum PMBUS_COMMANDS_Tag
{
    CMD_PAGE = 0x0,
    CMD_OPERATION = 0x01,
    CMD_ON_OFF_CONFIG = 0x02,
    CMD_CLEAR_FAULTS = 0x03,
    CMD_PHASE = 0x04,
    // 0x05-0x0F are reserved
    CMD_WRITE_PROTECT = 0x10,
```

8.2 配置应用程序特定的PMBUS命令

用户应用程序需要按照以下结构配置“已实现”的命令。请参见**pmbus_slave_user_commands.c**文件。

```
typedef struct PMBUS_CMD_LIST_Tag
{
    PMBUS_COMMANDS_T          cmd_code;
    PMBUS_BUS_PROTOCOL_T      wr_protocol;
    PMBUS_BUS_PROTOCOL_T      rd_protocol;
    uint8_t                   nRxBytesFrom_Host;
    uint8_t                   nTxBytesTo_Host;
    uint8_t                   *pTxToHostBuff;
} PMBUS_CMD_LIST_T;
```

```
const PMBUS_CMD_LIST_T gPMBUS_APPL_UserCMDList[] = {
// CMD Code             txn_w_type,             txn_r_type  nRxBytesFrom_Host nTxBytesTo_Host  pTxToHostBuff
// =========            ============            =========== ================  ==============  =============
{CMD_PAGE,               H_Wr_BYTE,              H_Rd_BYTE,          1,            1,          &gPMBUS_APPL_User_CMD_PAGE_Buffer },
{CMD_OPERATION,          H_Wr_BYTE,              H_Rd_BYTE,          1,            1,          &gPMBUS_APPL_User_OPERATION_Buffer},
```

在上面的示例中，

- CMD_PAGE支持BYTE READ和BYTE WRITE协议。
  - 因此，nRxBytesFrom_Host（从主机接收的字节数）和nTxBytesTo_Host（向主机发送的字节数）为1。
- 用户应用程序应为已实现的所有命令分配缓冲区。这些缓冲区的地址应在上表中提供。pTxToHostBuff → 指向缓冲区的指针。此缓冲区保存要发送给主机的值。（主机的READ命令）
  - 例如：1) PAGE命令支持WRITE和READ，因此 → gPMBUS_APPL_User_CMD_PAGE_Buffer为READ缓冲区。
    2) STATUS_WORD命令仅支持WORD READ协议，
    - wr_protocol = H_BUS_PROTOCOL_NA
    - rd_protocol = H_Rd_WORD
    - “READ”缓冲区（pTxToHostBuff）为0。接收到此命令后，协议栈自动将STATUS WORD发送给主机，而无需应用程序干预。

```
{CMD_STATUS_WORD,        H_BUS_PROTOCOL_NA,        H_Rd_WORD,        0,        2,        0},
```

注：用户应用程序需要使用“已实现/支持”的命令和关联的缓冲区来编译该表。

9.0 应用程序回调

从协议栈和应用程序使用以下结构进行交互。请参见pmbus_slave_driver.h。

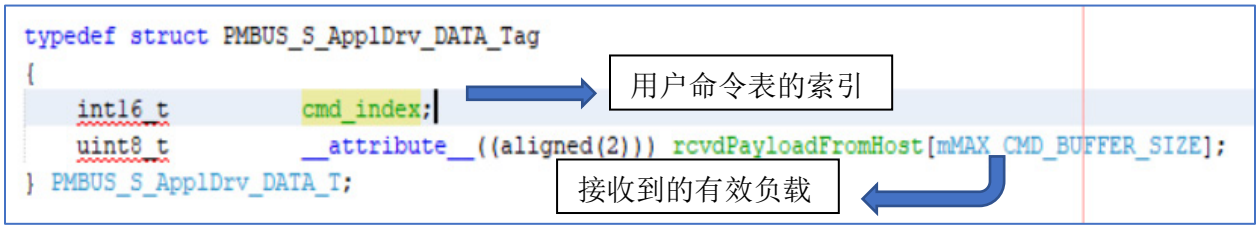
```
typedef struct PMBUS_S_ApplDrv_DATA_Tag
{
    int16_t          cmd_index;
    uint8_t          __attribute__((aligned(2))) rcvdPayloadFromHost[mMAX_CMD_BUFFER_SIZE];
} PMBUS_S_ApplDrv_DATA_T;
```

9.1 写协议

应用程序函数（回调处理程序——**PMBUS_DRV_Slave_Open**函数中提供的**ApplCallBackFuncPtr**）只有在成功接收（没有任何错误）之后才会被协议栈调用。

注：协议栈不处理数据转换/格式。用户应用程序需要处理数据转换/格式。

对于任何WRITE协议（包括PROCESS CALL写操作），从协议栈均提供



1) 命令索引。→ 命令列表结构的索引（指向接收到的命令）
2) 指向接收到的有效负载的指针。

复制接收到的有效负载的步骤。

1) 解码接收到的命令。
    - 使用PMBUS_S_APPLDrv_DATA_T结构中的cmd_index解码gPMBUS_APPL_UserCMDList结构中接收到的命令。

2) 用户应用程序应将接收到的有效负载复制到与接收到的命令（由主机发送）关联的“WRITE”缓冲区。
    - 从协议栈将接收到的有效负载填充/复制到PMBUS_S_APPLDrv_DATA_T结构中的rcvdPayloadFromHost缓冲区。

- **注：**对于BLOCK Write或Block Proc Write函数，接收到的有效负载的第一个字节表示**有效接收字节**数或有效负载的**字节数**。剩余字节为实际有效负载。

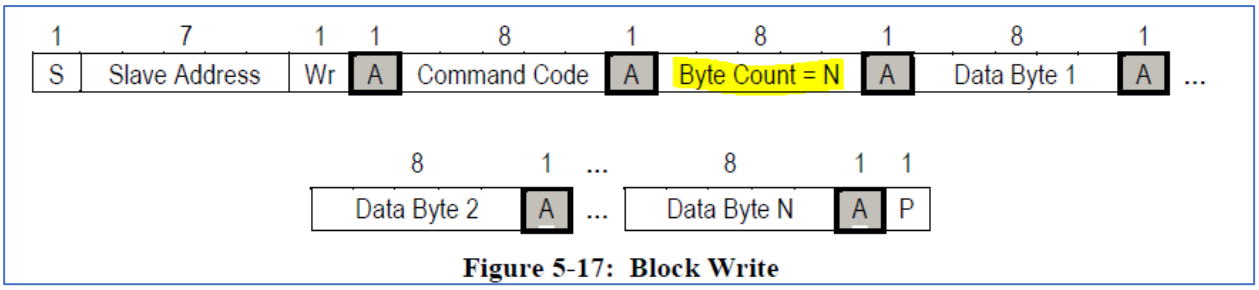


Figure 5-17: Block Write

- PMBus 1.1使用SMBus 1.1进行传输。请参见第1部分第5.1节的“SMBus，版本1.1”。根据SMBus 1.1，块传输允许的最大字节数为32。

> **7.5.7. Block Read/Write**
>
> The Block Write begins with a slave address and a write condition. After the command code the host issues a byte count which describes how many more bytes will follow in the message. If a slave had 20 bytes to send, the first byte would be the number 20 (14h), followed by the 20 bytes of data. The byte count may not be 0. A Block Read or Write is allowed to transfer a maximum of 32 data bytes.

例如：在演示应用程序中，PMBus_Device_Appl_CallBack为回调函数。请参见appl/slave_appl.c中该函数的实现。

```
void PMBus_Device_Appl_CallBack(PMBUS_S_ApplDrv_DATA_T appl_drv_data)
{
    // All errors are being handled by Slave driver itself. Like numofBytesToTx,
    // numOfBytesToRx etc. Hence no extra check is required here.
    Nop();
    uint8_t *srcData;
#ifdef mPRINT_DEBUG_MESSAGE
    UART_MessageAddToQueue((uint8_t *)"S_APPLCALLBACK\r\n", (uint16_t)strlen("S_APPLCALLBACK\r\n"));
#endif
    switch(gPMBUS_APPL_UserCMDList[appl_drv_data.cmd_index].cmd_code)
    {
        case CMD_PAGE:
        case CMD_OPERATION:
        case CMD_ON_OFF_CONFIG:
        case CMD_VOUT_COMMAND:
        case CMD_VOUT_TRIM:
        case CMD_MFR_ID:
        case CMD_EXT_MFR_SPECIFIC_1_BYTE:
        case CMD_EXT_MFR_SPECIFIC_2_WORD:

            if ((H_WrRd_BLOCK == gPMBUS_APPL_UserCMDList[appl_drv_data.cmd_index].cmd_txn_type) ||
                (H_Wr_BLOCK == gPMBUS_APPL_UserCMDList[appl_drv_data.cmd_index].cmd_txn_type))
            {
                srcData = appl_drv_data.rcvdPayloadFromHost;
                srcData++; // Point to Data.
                memcpy(
                (void*)gPMBUS_APPL_UserCMDList[appl_drv_data.cmd_index].pTxToHostBuff,
```

解码命令

将“接收到的有效负载”复制到用户应用程序缓冲区

9.2 读协议

所有读操作都是自动执行的。也就是说，协议栈将返回与命令关联的缓冲区的值，而无需应用程序干预。

注：对于Block RW Proc调用，在写操作之后，从协议栈将调用“应用程序回调”函数，以便应用程序对接收到的有效负载执行必要的操作。它也可以写入“ReadBuffer”。“ReadBuffer”中的该值将被发送到主机。

10.0 错误处理程序回调

如果主机和设备之间存在同步问题，则从协议栈将调用**PMBUS_DRV_Slave_Open**函数中的错误处理程序回调（**ErrorHandlerCallBackFuncPtr**）。同步问题的原因之一是，在与主机通信时，设备的I2C线先断开连接，随后又重新连接。

在这种情况下，建议使用以下函数关闭并重新打开PMBus设备：

- **PMBUS_DRV_Slave_Close**
- **PMBUS_DRV_Slave_Open**

```
void PMBus_Device_ErrHandler_CallBack(PMBUS_DRV_S_ERROR_CODES_T errCode)
{
    if (PMBUS_DRV_S_ERRCODE_RAISE_I2CINTR_PRI_LVL_DUE_TO_MANY_PENDING_INTRs == errCode)
    {
        // Slave I2C interrupt priority level to be raised, so that all pending
        // interrupts are serviced.

    }
    else if (PMBUS_DRV_S_NO_ERROR != errCode)
    {
        // Either Synchronization with Host or Too many interrupts pending issue.
        // Slave module should be closed and reinitialized.

        PMBUS_DRV_Slave_Close();
        // Disable Slave Interrupts.
        I2C_S_IF_Flag = 0;
        I2C_S_IE_Flag = 0;
        // Application should take necessary action here. !!!!

        // PMBUS Device/Slave module should be reopened.

        // Interrupts to be re-enabled.
        //I2C_S_IF_Flag = 0;
        //I2C_S_IE_Flag = 1;
        while(1); // Should be removed by application
    }
}
```

11.0 协议栈限制

1. SMBus协议——协议栈中不支持**Quick Command（快速命令）**。此协议仅用于存储容量较低的器件。
2. 协议栈中不处理“**Sending and Reading of too few bits**”（**发送和读取的位数太少**）的数据传输故障。这种情况通常是由于PMBus设备的热插拔引起的，插入的设备会短暂充当总线主设备。
3. 协议栈中不支持主机通知协议。
4. 不支持提醒响应地址（ARA）。发生故障时，设备会将SMALERT驱动为低电平。主机可以通过读取STATUS_WORD以循环轮转方式检查总线上的所有设备。成功读取STATUS_WORD后，主机可以发出CLEAR_FAULTS命令。
5. 协议栈中不支持任何命令的超时。用户应用程序可根据需要实现超时功能。
6. 如果主机在数据读取期间遇到“RESET”（复位），则从器件将保持数据线。主机负责产生其余的时钟脉冲，以便从器件释放数据线。之后主机可以启动停止条件并继续正常的总线通信。

12.0 存储器使用

闪存	~9K
RAM	~0.5 KB

13.0 疑难解答

I2C通信不工作：

- 验证是否已针对硬件连接配置了I2C实例（在config.h中）。
- 验证config_fuse.c中的备用或标准I2C引脚配置。
- 建议在SCLx和SDAx引脚上连接上拉电阻。如果使用了SMALERT引脚，则还应为其在外部连接上拉电阻。

PMBus从器件/设备与主机不同步：

- 如果从器件与I2C总线断开连接后又重新连接，则它将与主机不同步。在这种情况下，从协议栈将调用错误处理程序（**ErrorHandlerCallBackFuncPtr**）。
  协议栈将返回以下错误代码之一：
  - PMBUS_DRV_S_ERRCODE_SYNC_ISSUE_1_INVALID_STATE
  - PMBUS_DRV_S_ERRCODE_SYNC_ISSUE_2_INVALID_STATE
  - PMBUS_DRV_S_ERRCODE_SYNC_ISSUE_DUE_TO_PENDING_I2C_TX_RX_INTRs

用户应用程序需要先关闭模块，然后再将其打开。请参见appl/slave_appl.c中的PMBus_Device_ErrHandler_CallBack。

PMBus通信故障：

- PEC应在主机和设备中同时使能或禁止。如果不这样做，将导致通信故障和/或PEC故障。