1 基于 AC_CANFD_RS485 模块的 CAN 回环通信

章节导读

控制器局域网 CAN（Controller Area Network）属于现场总线的范畴，是一种有效支持分布式控制系统的串行通信网络。是由德国博世公司在 20 世纪 80 年代专门为汽车行业开发的一种串行通信总线。由于其高性能、高可靠性以及独特的设计而越来越受到人们的重视，被广泛应用于诸多领域。而且能够检测出产生的任何错误,所以我们学会使用 CAN 通信也是非常有意义的。在智多晶软件中提供有 CAN 控制器 IP 使用，本章我们就结合 CAN 控制器 IP 和 AC_CANFD_RS485 模块实现 CAN 通信，为了方便我们观测，还需要外接 CAN 调试器，实时检测发送和接收的数据。

1.1 系统的整体设计

在智多晶 SA5Z-50 FPGA 中通过 M33 硬核 MCU，通过 AHB 系统总线，AHB 跨时钟域模块，AHB 转 APB 模块，控制 A，B 两个 CAN 控制器 IP。整个系统的框图如下所示。

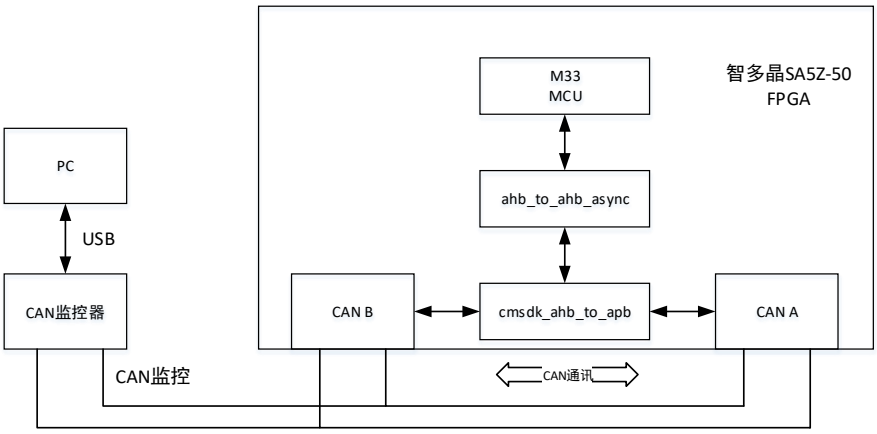


图 1-1 系统实现框图

具体实现的功能如下：

1. MCU 控制 CAN A 向 CAN B 发送数据+ID_A
2. CAN B 收到 A 发出的数据和 ID，中断给 MCU，MCU 读取信息
3. MCU 控制 CAN B 发出收到的数据+ID_B（与 ID_A 具有特征值的变化）

CAN 监控软件（BUSMASTER）通过 CAN 监控器（BUSMUST）持续监控

CAN 总线上的信息帧并持续刷新 A->B，B->A 的 CAN 帧信息

1.2 MCU 控制器配置

在 IP 管理器中找到 STAR 微控制器，首先是启用 GPIO 和 UART0。

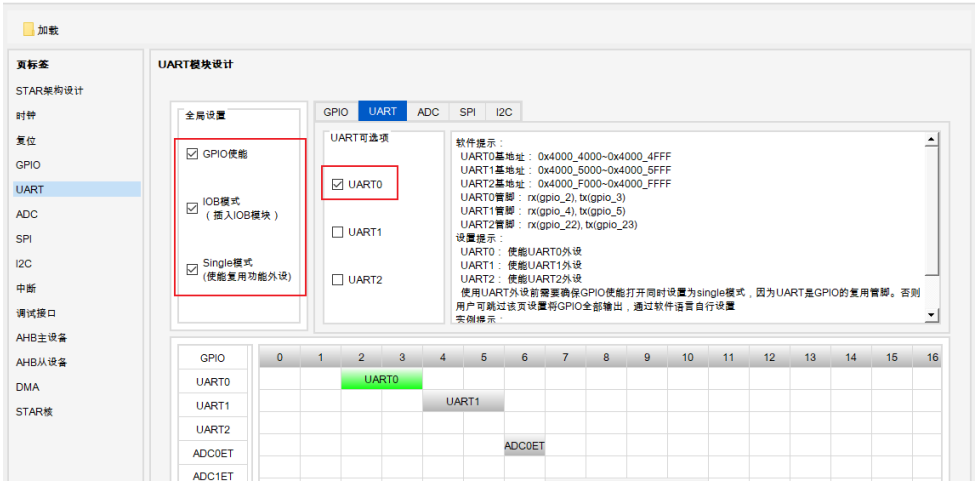


图 1-2 启用 UART0

然后启用外部中断，如下所示。

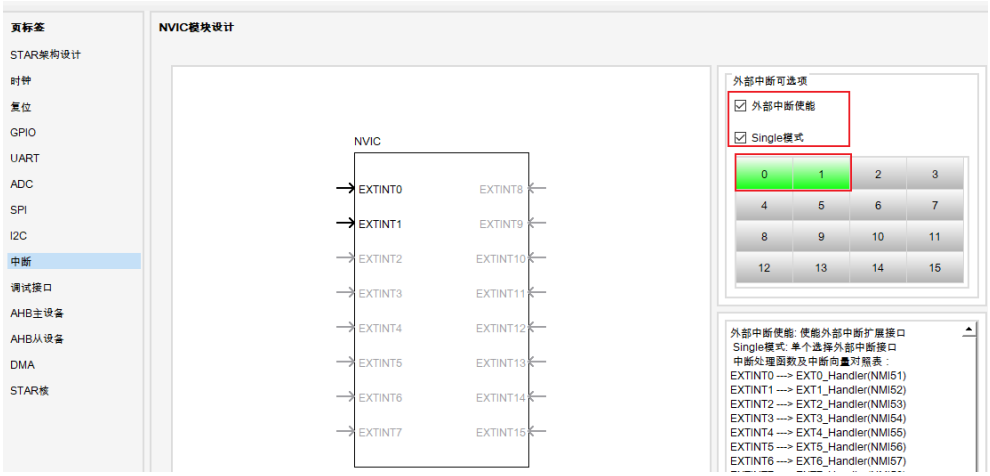


图 1-3 启用外部中断

启用 SW 调试接口，如下所示。

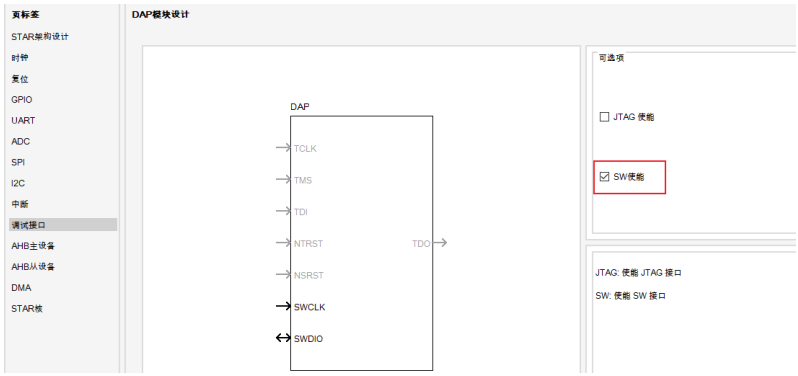


图 1-4 启用 SW 使能

使能 AHB 主设备的 Master0。

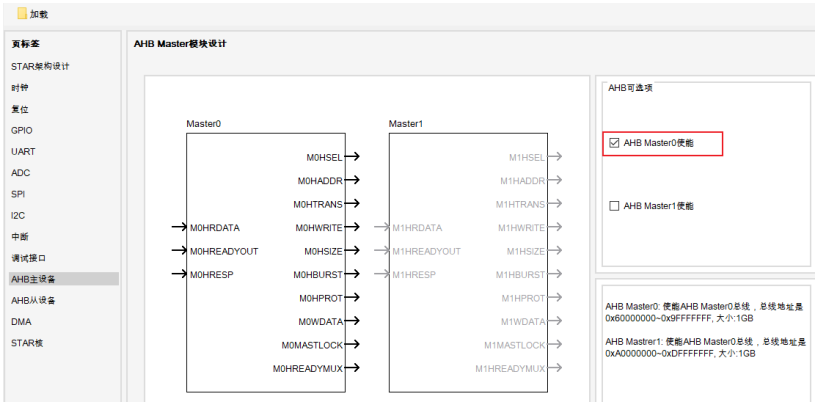


图 1-5 使能 AHB 主设备

1.3 ahb_to_ahb_async

ahb_to_ahb_async 模块就是调用的内部总线跨时钟适配器 Bus_CDC_Adapter IP，该 IP 支持 AHB_CDC 模式和 APB_CDC 模式两种跨时钟域的接口转换。在本次实验中 MCU 控制器输出的 AHB 信号的时钟是 200M，但是最终 CAN 控制器的 APB 总线的信号为 50M，所以这里就需要该模式实现跨时钟域的转换，转换完成之后再交由 AHB_to_APB 模块将 AHB 总线信号转换成 APB 信号，该 IP 配置界面如下图所示。

图 1-6 Bus_CDC_Adapter IP 配置界面

1.4 cmsdk_ahb_to_apb

前面我们提到过，MCU 的控制器输出的 AHB 总线的信号，CAN 控制器支持的是 APB 总线信号，所以这里我们就需要 cmsdk_ahb_to_apb 模块，用于在高级高性能总线（AHB）和高级外设总线（APB）之间进行协议转换。该模块的代码是官方提供的，网表形式的，具体代码我们看不到，我们使用的时候可以直接例化使用即可。

1.5 CAN 控制器

智多晶提供有 CAN Controller IP，该 IP 主要由 APB 接口模块（apb_intf）、寄存器模块（can_reg_map）、帧解析模块（can_packetizer）和位操作（can_bit）模块组成。系统框图如下所示。

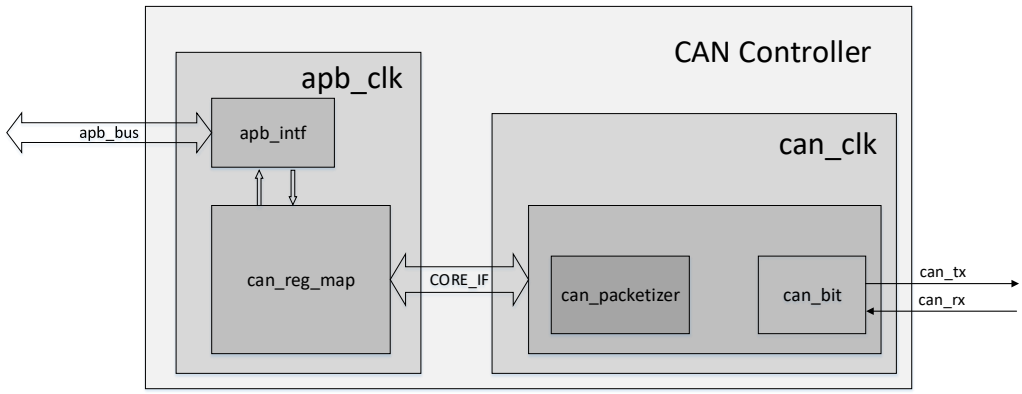


图 1-7 CAN 控制器系统框图

1.5.1 CAN 波特率计算

1.5.1.1 波特率与位组成

在 CAN 总线系统中，波特率的计算是一个关键步骤，它确保网络上的所有设备能够以相同的速率进行通信。波特率的计算涉及到几个关键参数，包括 CAN 控制器的时钟频率、分频因子、以及位时间的不同部分。

1. 时钟频率（Fclk）

CAN 控制器的时钟频率，这是 CAN 模块的输入时钟，通常来自于微控制器的主时钟。

2. 分频因子（Prescaler）

用于从主时钟频率中分频得到位时间计数器的时钟频率。分频因子可以增大位时间，使得 CAN 总线能够在较低的波特率下工作。

3. 位时间（Bit Time）

位时间一共由 4 个部分组成，具体如下图所示。

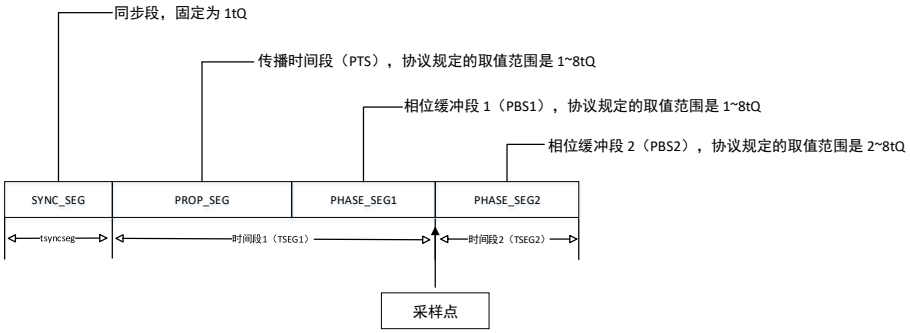


图 1-8 1bit 数据传输时段图

基于上述参数，CAN 波特率的计算公式如下：

波特率 = Fclk / (Prescaler * Bit Time)

其中，Bit Time = Sync Seg + Prop Seg + Phase Seg1 + Phase Seg2

CAN 总线位时间构成与时钟同步关系示意图如下所示。

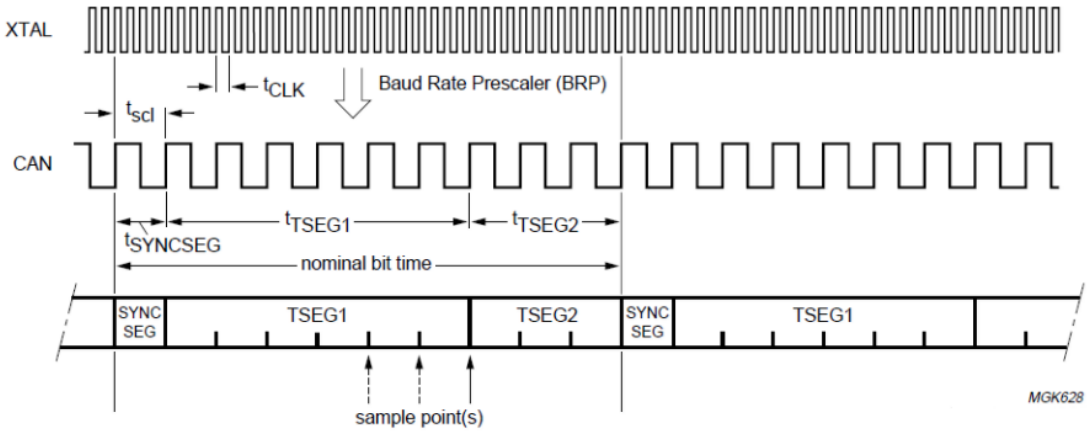


图 1-9 CAN 总线位时间构成与时钟同步关系示意图

相关参数定义如下：

Prescaler：分频很容易理解，控制器的时钟频率进行分频后会得到 CAN 的时钟，CAN 时钟的一个时间周期就是之前提到的最小时间段 tscl，也称作时间份额，代表的是 CAN 控制器工作时的最小时间单位。

SYNC_SEG：同步段用于同步总线上各个节点，固定长度为 1 个 tscl。其中应该有一个跳变沿。

PROP_SEG：传播段用于补偿信号通过网络和节点传播的物理延迟，长度

应能保证 2 倍的信号在总线的延迟，长度为 1 到 8 个 tscl。

PHASE_SEG：相位缓冲段用于补偿跳变沿的相位误差，相位缓冲段 1 的结尾是采样点位置。相位缓冲段 1 和 2 长度均为 1 到 8 个 tscl。

TSEG1 = PROP_SEG + PHASE_SEG1。

TSEG2 = PHASE_SEG2。

位时间 = SYNC_SEG + TSEG1 + TSEG2，长度为 8 到 25 个 tscl。

采样点 =（SYNC_SEG + TSEG1）÷位时间。

1.5.1.2 位同步原理

SJW（Resynchronization jump width），重同步宽度，协议规定的取值范围 1~4tQ。

CAN 总线使用位同步的方式来确保通信时序，以及对总线的电平进行正确采样。CAN 的同步分为硬同步和重同步。

硬同步在总线从 IDLE 状态进入 SOF 的第一个下降沿触发，再产生硬同步启动总线帧动作。总线状态机 IDLE 状态是一种低功耗状态，位同步模块并不工作，可由接收硬同步或主动发送行为启动位同步，总线状态机退出 IDLE 状态。在总线状态机退出至 IDLE 状态之前，即使帧接收/发送行为完成也不会再次进行硬同步。

重同步发生在总线帧收发过程中，因总线电平从“隐性”到“显性”的每一次跳变行为都会产生重同步，从而让整个 CAN 总线上各节点在传输过程中都能保持相当的位同步性。重同步的同步方式是通过在下一个 bit 周期增加 PHASE_SEG1 或减少 PHASE_SEG2 段长度来实现的。SJW (Resynchronization jump width)重同步宽度参数是对重同步可调宽度的限制。

根据 CAN Spec V2.0 的规定，SJW 允许的范围是 1~4。根据重同步的原理，要求 PHASE_SEG1>SJW，PHASE_SEG2>SJW。

1.5.1.3 BRP 参数

BRP (Baud rate prescaler)，预分频值，允许的范围是 1 ~ 255。BRP 用于将 APB 的时钟分频到 CAN 时钟。BRP 的值由基频（CAN_CLK）频率 Fclk 、波特率 Baudrate 和每个 Bit 的总节拍数 Total_tQ 决定。

$$\mathrm{BRP} = F_{clk}/(T_{otal\_tQ} * Baudrate)$$

例如：当 CAN_CLK 为 100MHz，CAN 的波特率为 1MHz，Total_tQ 值为 25，则 BRP 取值为 4。

1.5.1.4 PROP_SEG 的参数设置

根据 CAN spec V2.0，PROP_SEG 的意义在于补偿设备节点+连线的传播时延，取值算法为 2 倍的总线传播时延（线缆时延+节点驱动时延）。

时间长度根据设备性能和总线长度决定，通常总线单位延时可按 5.5ns/米估算。单个节点收/发延时按 75ns 估算。假定总线长度 12 米，则 PROP_SEG = 2 * (12 * 5.5 + 75 * 2) =432ns，然后根据上一步的 Total_t Q 值，计算出 t Q 的时间，再根据 tQ 和延时时间 432ns，计算 PROP_SEG 需要的节拍数。

若 tQ = 100ns，432ns 延时需要的节拍数 = 432/125，向上取整得到 PROP_SEG 取值是 5。

1.5.1.5 PHASE_SEG1 和 PHASE_SEG2 的参数设置

根据 Total_tQ 取值，将 Q 值减 1（SYNC_SEG），再减掉 PROP_SEG 的 t Q 节拍数，剩余的节拍数由 PROP_SEG1 和 PROP_SEG2 平分。如果剩余节拍是奇数，则 PROP_SEG1 比 PROP_SEG2 少一拍。

若 Total_tQ 取值 10，PROP_SEG 取值 5，则 PROP_SEG1=PROP_SEG2=2。

1.5.1.6 配置参考

表 1-1 典型配置参考

can_clk	Baudrate	Total_tQ	BRP	SJW	PROP_SEG	PHASE_SEG1	PHASE_SEG2
100MHz	1Mb/s	10	10	1	5	2	2
100MHz	500Kb/s	20	10	4	4	8	7
100MHz	50Kb/s	20	100	4	6	6	7

1.5.2 CAN 各类帧说明

CAN 网络通信是通过 5 种类型的帧进行的：

- 数据帧（Data frame）
- 遥控帧（Remote frame）
- 错误帧（Error frame）
- 过载帧（Overload frame）

- 帧间空间（Inter-frame space）

另外，数据帧和遥控帧有标准格式和扩展格式两种格式。标准格式有 11 个位的标识符（Identifier: 以下称 ID），扩展格式有 29 个位的 ID。

各种帧的用途如下表所示：

表 1-2 帧的类型及用途

帧类型	帧用途
数据帧（Data frame）	节点发送的包含 ID 和数据的帧，用于发送单元向接收单元传送数据的帧
遥控帧（Remote frame）	节点向网络上的其他节点发出的某个 ID 的数据请求，发送节点收到遥控帧后就可以发送相应 ID 的数据帧
错误帧（Error frame）	节点检测出错误时，向其他节点发送的通知错误的帧
过载帧（Overload frame）	接收单元未做好接收数据的准备时发送的帧，发送节点收到过载帧后可以暂缓发送数据帧
帧间空间（Inter-frame space）	用于将数据帧、遥控帧与前后的帧分隔开的帧

1. 数据帧

数据帧（Data frame）：用于发送单元向接收单元传送数据的帧。数据帧的构成如下图所示。

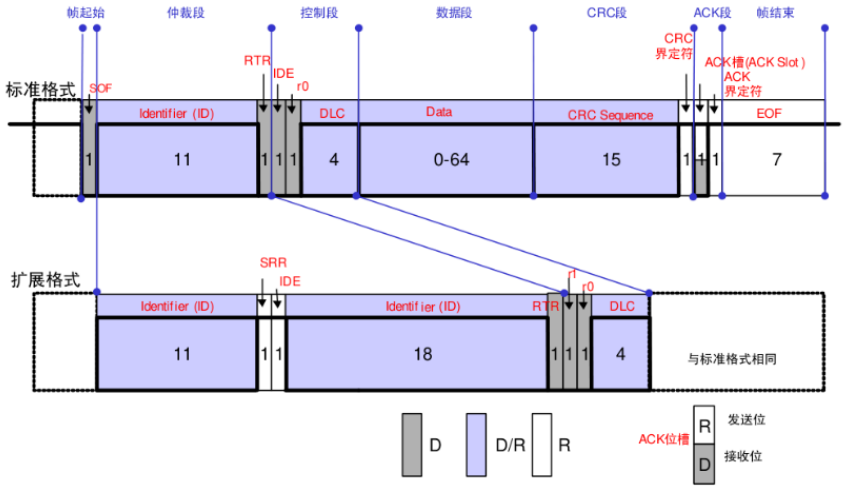


图 1-10 数据帧的构成

从上图可以看出，数据帧由 7 个段构成：

- 帧起始：表示数据帧开始的段。1 个位的显性位，总线上的电平具有显性电平和隐性电平两种。总线上执行逻辑上的线“与”时，显性电平的逻辑值为“0”，隐性电平为“1”。“显性”具有“优先”的意味，只要有一个单元输出显性电平，总线上即为显性电平。并且，“隐性”具有“包容”的意味，只有所有的单元都输出隐性电平，总线上才为音性电平。如下图所示。

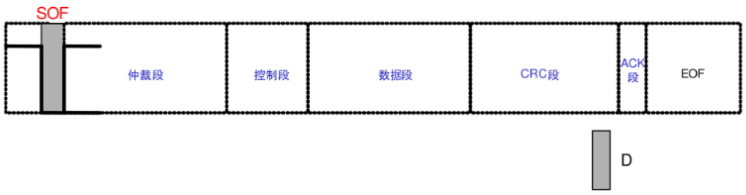


图 1-11 数据帧（帧起始）

- 仲裁段：表示该帧优先级的段。标准格式的 ID 有 11 个位。从 ID28 到 ID18 被依次发送。禁止高 7 位都为隐性（禁止设定：ID=1111111XXXX）。扩展格式的 ID 有 29 个位。基本 ID 从 ID28 到 ID18，扩展 ID 由 ID17 到 ID0 表示。基本 ID 和标准格式的 ID 相同。禁止高 7 位都为隐性。（禁止设定：基本 ID=1111111XXXX）。如下图所示。

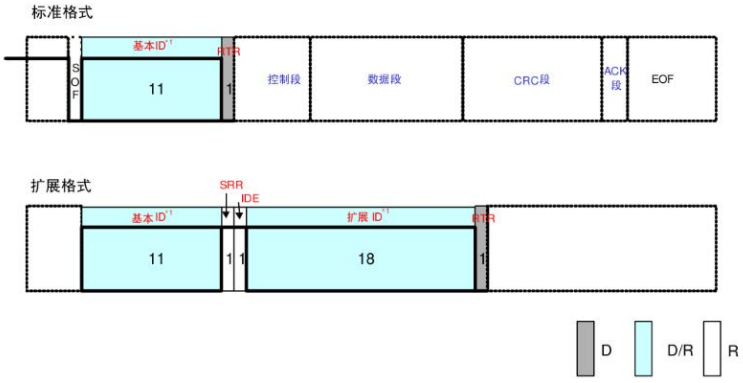


图 1-12 数据帧（仲裁段）

- 控制段：控制段由 6 个位构成，表示数据的字节数及保留位的段。如下图所示。

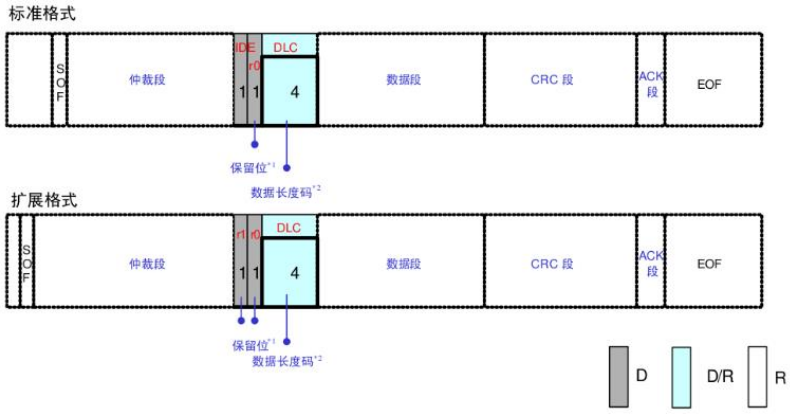


图 1-13

保留位（r0，r1），保留位必须全部以显性电平发送。但接收方可以接收显性、隐性及其任意组合的电平。数据长度码（DLC），数据长度码与数据字

节数的对应关系如下表所示。数据的字节数必须为 0~8 字节。但接收方对 DLC=9~15 的情况并不视为错误。

表 1-3 数据长度码和字节数的关系

数据字节数	数据长度码			
	DLC3	DLC2	DLC1	DLC0
0	D	D	D	D
1	D	D	D	R
2	D	D	R	D
3	D	D	R	R
4	D	R	D	D
5	D	R	D	R
6	D	R	R	D
7	D	R	R	R
8	R	D	D	D

“D”：显性电平 “R”：隐性电平

- 数据段：数据的内容，可发送 0~8 个字节的数据。从 MSB（最高位）开始输出。如下图所示。

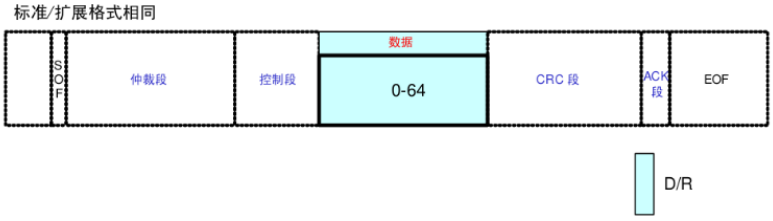


图 1-14 数据帧（数据段）

- CRC 段： CRC 段是检查帧传输错误的帧。由 15 个位的 CRC 顺序*1 和 1 个位的 CRC 界定符（用于分隔的位）构成。CRC 顺序是根据多项式生成的 CRC 值，CRC 的计算范围包括帧起始、仲裁段、控制段、数据段。接收方以同样的算法计算 CRC 值并进行比较，不一致时会通报错误。

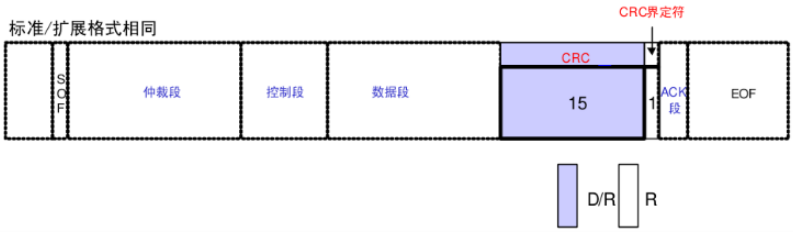


图 1-15 数据帧（CRC 段）

- ACK 段：ACK 段（Acknowledge Bit）用来确认是否正常接收。由 ACK 槽(ACK Slot)和 ACK 界定符 2 个位构成。如下所示。

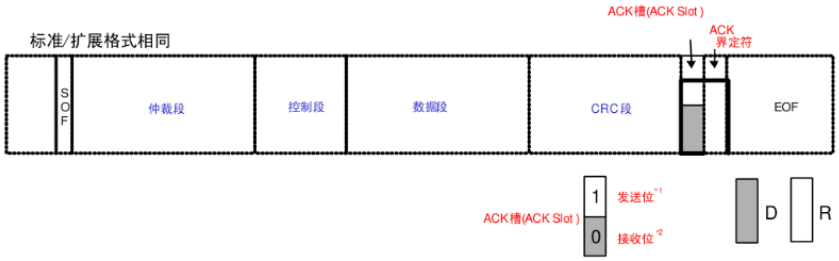


图 1-16 数据帧（ACK 段）

- 帧结束：帧结束是表示该该帧的结束的段。由 7 个位的隐性位构成。如下所示：

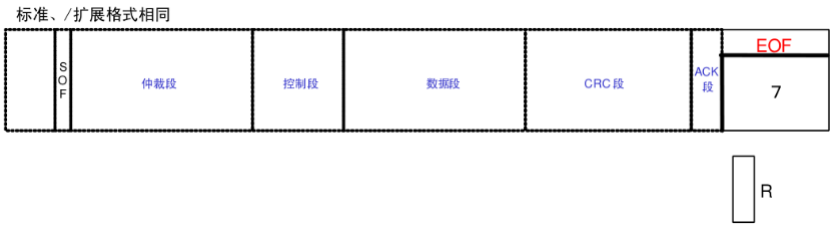


图 1-17 数据帧（帧结束）

2. 遥控帧

接收单元向发送单元请求发送数据所用的帧。遥控帧由 6 个段组成。遥控帧没有数据帧的数据段。遥控帧的构成如下所示。

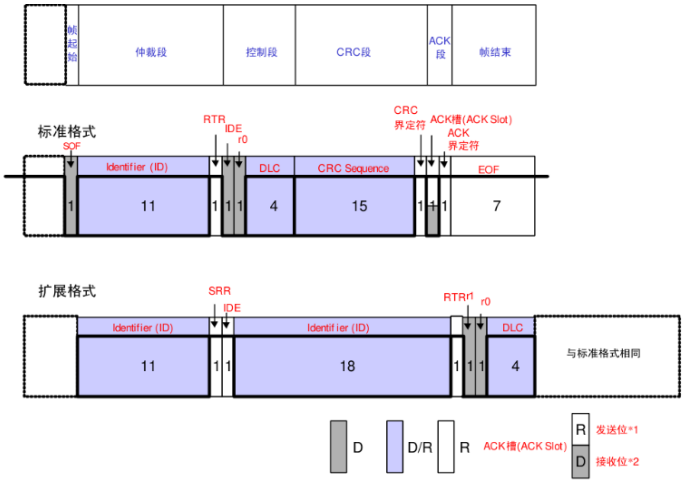


图 1-18 遥控帧的构成

从上图可以看出，与数据帧对比的不同：遥控帧的 RTR 位为隐性位，没有数据段。没有数据段的数据帧和遥控帧可通过 RTR 位区别开来。遥控帧的数据长度码以所请求数据帧的数据长度码表示。

3. 错误帧

用于在接收和发送消息时检测出错误通知错误的帧。错误帧由错误标志和错误界定符构成。错误帧如下图所示。错误标志包括主动错误标志和被动错误标志两种：主动错误标志：6 个位的显性位；被动错误标志：6 个位的隐性位。错误界定符由 8 个位的隐性位构成。

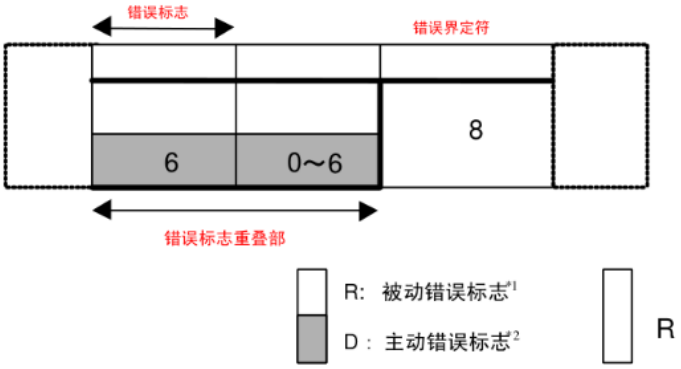


图 1-19 错误帧

4. 过载帧

过载帧是用于接收单元通知其尚未完成接收准备的帧。过载帧由过载标志和过载界定符构成。过载帧的构成如下图所示。过载标志：6个位的显性位，过载标志的构成与主动错误标志的构成相同；过载界定符：8个位的隐性位，过载界定符的构成与错误界定符的构成相同。

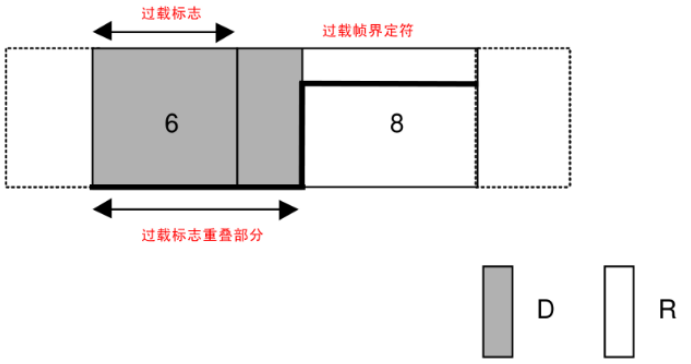


图 1-20 过载帧的构成

1.5.3 IP 配置

智多晶的 CAN 控制器的配置界面，如下图所示。

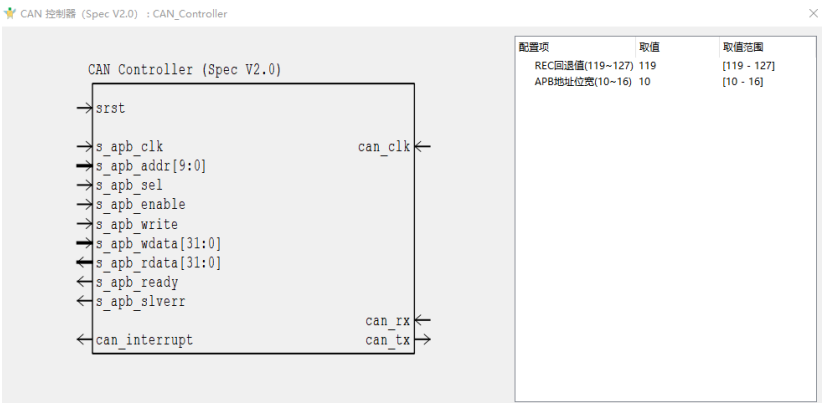


图 1-21 Can_Controller 配置界面

从上图可以看出，用户可配置 2 个参数，分别为“REC 回退值（119~127）”和“APB 地址位宽（10~16）”。

1.5.3.1 REC 回退值

根据 CAN 2.0 协议规定，当 CAN 设备有一个 TEC（发送错误计数器）和 REC（接收错误计数器）。当 CAN 设备成功监听到总线上一个无任何错误且成功 ACK 的消息包之后，如果 REC 当前值在 1~127（含）之间，REC 计数器减 1；如果 REC 当前值在 128~255（含）之间，则执行一个跳转到“回退值”的操作，即当前用户设置的“REC 回退值”。

协议中规定的回退值范围在 119（含）~127（含），本 IP 将 REC 回退值作为用户可配置的参数予以开放，用户可以自行设定。此参数在 IP 中的默认值为 119。

1.5.3.2 APB 地址位宽

用户选项为 10~16bit，用于确定 CAN Controller IP 中 APB 接口的 s_apb_addr 的位宽。关于 SoC 访问设备上某个寄存器时的地址和地址运算，基地址=APB 外设扩展地址+设备基地址。总线上的寄存器访问地址由基地址和寄存器偏移地址组成。根据用户实际选择的接口外设而决定。

1. APB 外设扩展地址

根据用户实际选择的接口外设而决定，在本 demo 中使用的 APB 接口外设起始地址为 0x00040000。

2. 设备基地址

用于在同一个 APB 外设地址段挂载多个模块时使用，由于 APB 外设扩展地址可寻址空间位宽为 16bit，则可设置的设备基地址位宽为：32-16-寄存器偏移地址位宽。若寄存器偏移地址位宽设置为 10，则设备基地址段为 6bit，若寄存器偏移地址位宽设置为 12，则设备基地址段为 4bit。根据用户挂载 APB 从设备的应用配置，用户可决定每一个 APB 从设备的实际基地址，即：{APB 外设扩展地址，设备基地址}这样的地址拼接。

对于 CAN Controller IP 外设而言，寄存器偏移地址之外的地址解析由 s_apb_sel 信号实现。比如说，用户系统中 CAN Controller IP 的基地址配置为 32’hF8157000，并且高 20bit 作为设备基地址（BASE_ADDRESS）为 20’hF8157，当逻辑判断用户通过 MCU 访问 APB 总线上的高 20bit 地址等于 20’hF8157 时，对 s_apb_sel 置“1”，激活当前设备在其 APB 寄存器偏移地址部分的解析。

1.5.4 接口说明

表 1-4 CAN Controller IP 信号说明表

端口名	方向	数据位宽	时钟域	描述
srst	input	1bit	s_apb_clk	复位信号（同步复位，高有效）
can_clk	input	1bit	can_clk	CAN Controller IP 工作时钟
s_apb_clk	input	1bit	s_apb_clk	APB 接口时钟
s_apb_write	input	1bit	s_apb_clk	apb_slave 读写信号 1：apb 写指令 0：apb 读指令
s_apb_sel	input	1bit	s_apb_clk	apb_slave 选通信号 1：apb 选中 0：apb 未选中
s_apb_enable	input	1bit	s_apb_clk	apb_slave 选通信号 1：apb 选中 0：apb 未选中
s_apb_addr	input	10~16bit	s_apb_clk	apb_slave 地址，可通过参数配置位宽
s_apb_wdata	input	32bit	s_apb_clk	apb_slave 写入数据
s_apb_rdata	output	32bit	s_apb_clk	apb_slave 读出数据
s_apb_ready	output	1bit	s_apb_clk	apb_slave 读写握手信号
s_apb_slverr	output	1bit	s_apb_clk	apb_slave 协议报错信号，功能关闭，强制输出 0
can_interrupt	output	1bit	s_apb_clk	中断信号，包含了“收/发包完成”，“can 节点 error 状态”，“5 个 error 状态”共 9 个中断源。 1：表示当前存在中断 0：当前无中断
can_tx	output	1bit	can_clk	can 数据发送通道

can_rx	input	1bit	can_clk	can 数据接收通道

1.6 apb_mux 模块

我们本次实验用到了两个 CAN 模块，一个用于发送，一个用于接收，也就是说我们有两个 APB 从设备，这里的 apb_mux 模块实现了一个 APB (Advanced Peripheral Bus) 总线多路复用器，用于将单个 APB 主设备连接到多个 APB 从设备。多路复用器根据地址匹配决定将主设备的请求路由到哪个从设备，并将响应返回给主设备。

首先在模块中我们定义多个参数来配置总线宽度和地址范围，如下所示：

```
parameter APB_DATA_BYTE_NUM = 4;      // 数据总线字节数
parameter APB_DATA_WIDTH = APB_DATA_BYTE_NUM*8;  // 数据总线宽度(32 位)
parameter APB_ADDR_WIDTH = 16;        // 地址总线宽度
parameter APB_SUB_ADR_WIDTH = 8;      // 子地址宽度
parameter DEV0_BASEADDR = 16'h0000;   // 第一个从设备基地址
parameter DEV1_BASEADDR = 16'h1000;   // 第二个从设备基地址
```

然后从 APB 中提取高 8 位作为基地址部分，将基地址与两个从设备的基地址比较，生成匹配信号 BASEADR_match，指示此时是哪个从设备，代码如下所示：

```
wire [APB_ADDR_WIDTH-APB_SUB_ADR_WIDTH-1:0] apb_base_addr;
assign apb_base_addr = s_apb_addr[APB_ADDR_WIDTH-1:APB_SUB_ADR_WIDTH];

wire [1:0] BASEADR_match;
assign BASEADR_match[0] = (apb_base_addr==DEV0_BASEADDR[APB_ADDR_WIDTH-
1:APB_SUB_ADR_WIDTH])? 1'b1 : 1'b0;
assign BASEADR_match[1] = (apb_base_addr==DEV1_BASEADDR[APB_ADDR_WIDTH-
1:APB_SUB_ADR_WIDTH])? 1'b1 : 1'b0;
```

选择信号生成：只有当主设备选择信号 s_apb_sel 有效且地址匹配某个从设备时，对应的从设备选择信号才会有效，代码如下所示：

```
assign s_apb_sel_0 = s_apb_sel & BASEADR_match[0];
assign s_apb_sel_1 = s_apb_sel & BASEADR_match[1];
```

控制信号和数据直接复制到两个从设备接口，虽然所有信号都被复制到两个从设备，但只有被选中的从设备才会响应，因为 SEL 信号无效的从设备会忽略这些输入，代码如下所示：

```
assign s_apb_enable_0 = s_apb_enable;
assign s_apb_addr_0 = s_apb_addr;
assign s_apb_prot_0 = s_apb_prot;
assign s_apb_write_0 = s_apb_write;
assign s_apb_wdata_0 = s_apb_wdata;
assign s_apb_strb_0 = s_apb_strb;
```

```
assign s_apb_enable_1 = s_apb_enable;
assign s_apb_addr_1 = s_apb_addr;
assign s_apb_prot_1 = s_apb_prot;
assign s_apb_write_1 = s_apb_write;
assign s_apb_wdata_1 = s_apb_wdata;
assign s_apb_strb_1 = s_apb_strb;
```

从设备的响应信号通过条件选择后合并，只有被选中的从设备的响应才会被传递给主设备，未被选中的从设备的响应被置为默认值（READY=0, SLVERR=0, RDATA=0），最终主设备看到的 READY、SLVERR 和 RDATA 信号根据选择结果合并后的结果，代码如下所示：

```
assign s_apb_ready_0_use = (s_apb_sel_0) ? s_apb_ready_0 : 1'b0;
assign s_apb_ready_1_use = (s_apb_sel_1) ? s_apb_ready_1 : 1'b0;
assign s_apb_slverr_0_use = (s_apb_sel_0) ? s_apb_slverr_0 : 1'b0;
assign s_apb_slverr_1_use = (s_apb_sel_1) ? s_apb_slverr_1 : 1'b0;
assign s_apb_rdata_0_use=(s_apb_sel_0)?s_apb_rdata_0:
{(APB_DATA_WIDTH){1'b0}};
assign s_apb_rdata_1_use = (s_apb_sel_1) ? s_apb_rdata_1 :
{(APB_DATA_WIDTH){1'b0}};

assign s_apb_ready = s_apb_ready_0_use | s_apb_ready_1_use;
assign s_apb_slverr = s_apb_slverr_0_use | s_apb_slverr_1_use;
assign s_apb_rdata = s_apb_rdata_0_use | s_apb_rdata_1_use;
```

完整的代码请自行查看对应工程。

1.7 创建顶层模块

然后创建顶层模块，将所有模块进行例化，完整代码我们这里就不进行展示了，请自行查看工程中提供的源码，创建完成之后，进行综合。

1.8 物理约束

本次实验的物理约束如下图所示。

Status	Ports	Direction	Location	IO_TYPE	PULLMODE	CLAMP
Enabled	clkin_mcu	→ INPUT	K15 bank2	LVCMOS33	NONE	ON
Enabled	io_asyncResetn	→ INPUT	H14 bank2	LVCMOS33	NONE	ON
Enabled	uart_rxd	→ INPUT	U16 bank3	LVCMOS33	NONE	ON
Enabled	can0_rx	→ INPUT	A18 bank2	LVCMOS33	NONE	ON
Enabled	can1_rx	→ INPUT	M17 bank3	LVCMOS33	NONE	ON
Enabled	SWCLK	→ INPUT	P17 bank3	LVCMOS33	NONE	ON
Enabled	work_led	← OUTPUT	D12 bank2	LVCMOS33	NONE	OFF
Enabled	uart_txd	← OUTPUT	U13 bank3	LVCMOS33	NONE	OFF
Enabled	can0_tx	← OUTPUT	J17 bank2	LVCMOS33	NONE	OFF
Enabled	can1_tx	← OUTPUT	N17 bank3	LVCMOS33	NONE	OFF
Enabled	SWDIO	↔ INOUT	R15 bank3	LVCMOS33	NONE	ON

图 1-22 物理约束

约束完成之后，随后进行布局布线，并生成比特流。

1.9 MDK 工程建立

在 MDK 工程，我们需要对中断进行初始化，其中中断 extint0 对应 CAN0 的中断，中断 extint1 对应 CAN1 的中断，并对 CAN 回环进行初始化，并间隔 1S 发送一帧数据，main.c 的代码如下所示：

```
#include <stdio.h>
#include <inttypes.h>
#include "STAR.h"
#include "STAR_gpio.h"
#include "uart.h"
#include "ahb.h"
#include "extint.h"
#include "misc.h"
#include "can_loopback.h"
#include "can_api.h"
extern u8 send_flag;
extern CAN_INT_SRC  can_int_src1;
int main(void)
{

    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);
    extint0_init();     //init interrrput 0
    extint1_init();     //init interrrput 1
    init_can_loopback();
    u16 num = 0;
    u8 time_cnt = 0;
    send_flag = 0;
    while(1)
    {
        can_auto_test(num,&can_int_src1);
```

```
            time_cnt ++;
            if(time_cnt >= 50)
            {
                time_cnt = 0;
                send_flag = 1;
                num++;
            }
        }
}
```

1.9.1 CAN 相关寄存器

在进行 CAN 相关代码的设置之前，我们得需要对 CAN 控制器的相关寄存器进行简要说明，代码如下所示：

表 1-5 CAN 控制寄存器详解

Offset Address	Reg Name	Bits	Field Name	Access	Description
0x000	STATUS_REG	[31:4]	reserved		
		[3]	CAN_READY	RO	1:CAN controller ready to use. 0:CAN controller not ready
		[2:0]	CAN_STATUS	RO	CAN FSM status. 000: IDLE 100: Normal Frame 010:Overload Frame 001: Error Frame Else: Reserved
0x004	INT_CTRL_REG	[31]	TX_FNSH_INT_EN	RW	1:TX_FNSH interrupt enable. 0:TX_FNSH interrupt disable.
		[30]	RX_FNSH_INT_EN	RW	1:RX_FNSH interrupt enable. 0:RX_FNSH interrupt disable
		[29:8]	reserved		
		[7]	FLAG_BUS_OFF_INT_EN	RW	1:FLAG_BUS_OFF interrupt enable. 0:FLAG_BUS_OFF interrupt disable.
		[6]	FLAG_PASSIVE_ERR_INT_EN	RW	1: FLAG_PASSIVE_ERR interrupt enable. 0: FLAG_PASSIVE_ERR interrupt disable.
		[5]	reserved		
		[4]	ERR_STUFF_INT_EN	RW	1: ERR_STUFF interrupt enable. 0: ERR_STUFF interrupt disable.
		[3]	ERR_FORM_INT_EN	RW	1: ERR_FORM interrupt enable. 0: ERR_FORM interrupt disable.
		[2]	ERR_ACK_INT_EN	RW	1: ERR_ACK interrupt enable. 0: ERR_ACK interrupt disable.
		[1]	ERR_BIT_INT_EN	RW	1: ERR_BIT interrupt enable. 0: ERR_BIT interrupt disable.
		[0]	ERR_CRC_INT_EN	RW	1: ERR_CRC interrupt enable 0: ERR_CRC interrupt disable
0x008	INT_SRC_REG	[31]	TX_FNSH	RC	1: TX packet transmit completed. 0: TX packet transmit not completed. Once 0x008 (INT_SRC_REG) been read from APB, TX_FNSH bit will be cleared to 0.
		[30]	RX_FNSH	RC	1: RX packet & data received. 0: RX packet & data not received. Once 0x008 (INT_SRC_REG) been read from APB, RX_FNSH bit will be cleared to 0.
		[29:24]	reserved		
		[23:16]	REC	RO	Value of Receive error counter, 8bit unsigned decimal.
		[15:8]	TEC	RO	Value of Transmit error counter, 8bit unsigned decimal.
		[7]	FLAG_BUS_OFF	RO	1: Entering Bus-off flag 0: Not Entering Bus-off flag

		[6]	FLAG_PASSIVE_ERR	RO	1: Entering Passive-error flag 0: Not Entering Passive-error flag
		[5]	reserved		
		[4]	ERR_STUFF	RO	Stuff Error indicator 1: Stuff Error detected 0: No Stuff Error
		[3]	ERR_FORM	RO	Form Error indicator 1: Form Error detected 0: No Form Error
		[2]	ERR_ACK	RO	Acknowledgment Error indicator 1:ACK Error detected 0: NoACK Error
		[1]	ERR_BIT	RO	Bit Error indicator 1: Bit Error detected 0: No Bit Error
		[0]	ERR_CRC	RO	CRC Error indicator 1: CRC Error detected 0: No CRC Error
0x00C	CONFIG _REG	[31]	CFG_RESET	WC	Software reset to CAN controller. Write 1 to generate soft reset of CAN controller. CFG_RESET bit is always auto-cleared by itself, user does not have to write 0 to CFG_RESET bit.
		[30]	CFG_OVLD	RW	Request for overload frame,to occupy can bus Write 1 to activate active-overload from master, which force CAN controller keep generating the overload frame. Write 0 to deactivate active-overload and release CAN bus occupation. (default=0)
		[29]	CFG_ERR_CLR	WC	Clear error reports in INT_SRC_REG[4:0]。 Wirte 1 to clear ERR_STUFF,ERR_FORM, ERR_ACK, ERR_BIT,ERR_CRC bits in 0x008(INT_SRC_REG) to 0.
		[28:24]	reserved		
		[23:16]	CFG_BRP	RW	Baud rate prescaler value setting. (1~255)
		[15:12]	CFG_SJW	RW	Re-synchronization jump width value setting. (1~4)
		[11:8]	CFG_PTS	RW	Propagation Time Segment value setting. (1~8)
		[7:4]	CFG_PBS1	RW	Define PHASE_SEG1 t Q value setting. (1~8)
		[3:0]	CFG_PBS2	RW	Define PHASE_SEG2 t Q value setting.(2~8)
0x010 ~0x0FC	Reserved				
0x100	TX_PKT_REG 1	[31]	TX_PKT_IDE	RW	Identifier bit of extend frame (TX frame). 1: The TX frame is extended format (29bit identifier) 0: The TX frame is standard format (11bit identifier)
		[30:29]	reserved		
		[28:0]	TX_PKT_ID	RW	In standard format (11 bit identifier) frame, can controller use TX_PKT_ID[28:18] as the 11 bit identifier value. In extended format (29 bit identifier) frame, can controller use all 29 bits of TX_PKT_ID as the 29 bit identifier value. TX_PKT_ID[28:18]:TX Standard packet identifier TX_PKT_ID[28:0]: TX Extend packet identifier
0x104	TX_PKT_REG 2	[31:6]	reserved		
		[5]	TX_PKT_RTR	RW	Remote transmission request indicator (TX frame). 1: The TX frame is Remote frame. 0: The TX frame is Data frame.
		[4]	TX_PKT_REQ	WC	Transmit request.auto clear when TX packet finish. 1: Write 1 to request for sending a frame. 0: No TX request or the last request have been executed.
		[3:0]	TX_PKT_DLC	RW	Data Length Code of the TX frame. (0~8)
0x108	TX_DAT A_REG1	[31:24]	TX_DATA_BYTE4	RW	TX Data byte 4 of the message.
		[23:16]	TX_DATA_BYTE3	RW	TX Data byte 3 of the message.
		[15:8]	TX_DATA_BYTE2	RW	TX Data byte 2 of the message.
		[7:0]	TX_DATA_BYTE1	RW	TX Data byte 1 of the message.
0x10C	TX_DAT A_REG2	[31:24]	TX_DATA_BYTE8	RW	TX Data byte 8 of the message.
		[23:16]	TX_DATA_BYTE7	RW	TX Data byte 7 of the message.
		[15:8]	TX_DATA_BYTE6	RW	TX Data byte 6 of the message.
		[7:0]	TX_DATA_BYTE5	RW	TX Data byte 5 of the message.
0x200	RX_PKT	[31]	RX_PKT_IDE	RO	Identifier bit of extend frame (received

	_REG1				frame). 1: The frame received is extended format (29bit identifier) 0: The frame received is standard format (11bit identifier)
		[30:29]	reserved		
		[28:0]	RX_PKT_ID	RO	In standard format (11 bit identifier) frame, can controller use RX_PKT_ID[28:18] as the 11 bit identifier value. In extended format (29 bit identifier) frame, can controller use all 29 bits of RX_PKT_ID as the 29 bit identifier value. RX_PKT_IDE bit value is needed for users to determine the useful bits of the RX_PKT_ID register. RX_PKT_ID[28:18]: RX Standard packet identifier RX_PKT_ID[28:0]: RX Extend packet identifier
		[31:6]	reserved		
0x204	RX_PKT _REG2	[5]	RX_PKT_RTR	RO	Remote transmission request indicator(RX frame). 1: The received frame is Remote frame. 0: The received frame is Data frame.
		[4]	reserved		
		[3:0]	RX_PKT_DLC	RO	Data Length Code of the received frame
0x208	RX_DAT A_REG1	[31:24]	RX_DATA_BYTE4	RO	RX Data byte 4 of the message.
		[23:16]	RX_DATA_BYTE3	RO	RX Data byte 3 of the message.
		[15:8]	RX_DATA_BYTE2	RO	RX Data byte 2 of the message.
		[7:0]	RX_DATA_BYTE1	RO	RX Data byte 1 of the message.
0x20C	RX_DAT A_REG2	[31:24]	RX_DATA_BYTE8	RO	RX Data byte 8 of the message.
		[23:16]	RX_DATA_BYTE7	RO	RX Data byte 7 of the message.
		[15:8]	RX_DATA_BYTE6	RO	RX Data byte 6 of the message.
		[7:0]	RX_DATA_BYTE5	RO	RX Data byte 5 of the message.

1.9.2 CAN 控制器初始化

初始化第一步就是配置时序参数，也就是配置 CAN 控制器的波特率，根据表 1-1 典型配置参考中的内容，将波特率设置成 1Mb/s，也就是 BRP 为 10，SJW 为 1，PROP_SEG 为 5，PHASE_SEG1 和 PHASE_SEG2 为 2，通过 can_bit_timing_set 函数将对应指针中，然后通过 can_rate_set 函数将数据写入对应寄存器中，根据表 1-5 CAN 控制寄存器详解中的内容，需要写入的寄存器为 CONFIG_REG，代码如下所示：

```
/* 功能：can 位时序值赋值
 * 参数：bit_timing 设备位时序参数结构体的地址指针
 */
void can_bit_timing_set(CAN_BIT_TIMING *bit_timing)
{
    //100Mhz(apb_clk),can_rate=1Mb/s
    bit_timing->brp     = 10;
    bit_timing->sjw     = 1;
    bit_timing->pts     = 5;
    bit_timing->pbs1    = 2;
    bit_timing->pbs2    = 2;
}

/* 功能：CAN 速率配置，位时序配置，相当于 can 的初始化
 * 参数：dev_addr 设备基地址（16bit，其中对应寄存器偏移地址的低位 bit 全部为 0）
```

```
 * 参数：bit_timing 设备位时序参数结构体指针
 */
u8 can_rate_set(u16 dev_addr,CAN_BIT_TIMING *bit_timing)
{
    u32 temp = apb3_read2(APB0+dev_addr,CONFIG_REG) & 0x40000000;
    u32 timing = ((u32)bit_timing->brp<<16) +
                 ((u32)bit_timing->sjw<<12) +
                 ((u32)bit_timing->pts<<8) +
                 ((u32)bit_timing->pbs1<<4) +
                 (u32)bit_timing->pbs2;

    apb3_write2(APB0+dev_addr,CONFIG_REG,timing | temp);
    return 0;
}
```

然后是中断配置，根据表 1-5 CAN 控制寄存器详解中的内容就是配置 INT_CTRL_REG 寄存器的内容，通过 set_int_en 函数进行中断使能配置，代码如下所示：

```
u8 set_int_en(u16 dev_addr,u32 int_en)
{
    apb3_write2(APB0+dev_addr,INT_CTRL_REG,(u32)int_en);
    apb3_read2(APB0+dev_addr,INT_SRC_REG);
    return 0;
}
```

最终在 init_can_loopback 函数中配置 DEV0 和 DEV1 的波特率为 1Mb/s，并且启动 DEV0 的错误中断，DEV1 的接收中断。代码如下所示：

```
u8 init_can_loopback()
{
    //dev0,1mb/s
    can_bit_timing_set(&can_bit_timing0);            //配置位时序参数
    can_rate_set(DEV0_BASEADDR,&can_bit_timing0);
    //dev1,1mb/s
    can_bit_timing_set(&can_bit_timing1);            //配置位时序参数
    can_rate_set(DEV1_BASEADDR,&can_bit_timing1);

    set_int_en(DEV0_BASEADDR,0xdf);          //dev0 启动错误中断
    set_int_en(DEV1_BASEADDR,0x40000000);    //dev1 启动接收中断
    return 0;
}
```

1.9.3 发送一帧数据函数

帧的发送主要涉及 0x100~0x10C 的 4 个寄存器，寄存器 0x100 为发送帧的 standard ID(base ID)、extended ID 和 IDE BIT3 个字段的配置寄存器。寄存器 0x108 和 0x10C 为发送帧的 DATA 段的配置寄存器，具体 byte 请参考寄存器表

中的内容。寄存器 0x104 为 TX_PKT_REQ （发送请求）和 TX_PKT_RTR、TX_PKT_DLC 字段的配置寄存器。

发送一帧数据函数代码如下所示：

```
u8 can_send_msg(u16 dev_addr,CAN_MSG *txMsg)
{
    u32 tx_pkt_reg1  = ((u32)(txMsg->std_id<<21)) +
                       ((u32)(txMsg->ext_id<<3)) +
                       ((u32)(txMsg->ide<<2));
    u32 tx_pkt_reg2  = ((u32)(txMsg->rtr<<5)) +
                       0x10 +                    //tx_pkt_req
                       ((u32)txMsg->dlc);
    u32 tx_data_reg1 = txMsg->data1;
    u32 tx_data_reg2 = txMsg->data2;

    //write reg
    apb3_write2(APB0+dev_addr,TX_PKT_REG1,tx_pkt_reg1);//ID+IDE
    apb3_write2(APB0+dev_addr,TX_DATA_REG1,tx_data_reg1);//DATAL
    apb3_write2(APB0+dev_addr,TX_DATA_REG2,tx_data_reg2);//DATAH
    apb3_write2(APB0+dev_addr,TX_PKT_REG2,tx_pkt_reg2);//TXRQ+RTR+DLC
    return 0;
}
```

1.9.4 接收一帧数据函数

帧的接收主要涉及 0x200~0x20C 的 4 个寄存器和中断源寄存器。寄存器 0x200 为接收帧的 standard ID(base ID)、extended ID 和 IDE BIT3 个字段的存储寄存器。寄存器 0x208 和 0x20C 为接收帧的 DATA 段的存储寄存器，具体 byte 请参考寄存器表。寄存器 0x204 为 RX_PKT_RTR、RX_PKT_DLC 字段的存储寄存器。接收一帧数据的函数代码如下所示：

```
u8 can_rec_msg(u16 dev_addr,CAN_MSG *rxMsg)
{

    reg[0] = apb3_read2(APB0+dev_addr,RX_PKT_REG1);
    reg[1] = apb3_read2(APB0+dev_addr,RX_PKT_REG2);
    reg[2] = apb3_read2(APB0+dev_addr,RX_DATA_REG1);
    reg[3] = apb3_read2(APB0+dev_addr,RX_DATA_REG2);

#ifdef DBG
    printf("CAN:RECV:%08X,%08X,%08X,%08X\r\n",reg[0],reg[1],reg[2],reg
[3]);
#endif
    rxMsg->std_id = (u16)(reg[0] >>21) & 0x7ff;
    rxMsg->ext_id = (u32)(reg[0] >>3) & 0x3ffff;
```

```
    rxMsg->ide    = (u8 )(reg[0] >>2) & 0x01;
    rxMsg->rtr    = (u8 )(reg[1] >>5 ) & 0x01;
    rxMsg->dlc    = (u8 )(reg[1]     ) & 0x0f;
    rxMsg->data1  = reg[2];
    rxMsg->data2  = reg[3];
    return 0;
}
```

1.9.5 回环数据函数实现

当间隔 1S 时间到了之后，设备 0 发送数据，ID 为 1 依次增加，发送的数据 data1 为 ID，data2 为 ID 加 1，然后如果设备 B 接收到，就回发，然后将数据再发送出去，代码如下所示：

```
u8 can_auto_test(u16 num,CAN_INT_SRC *int_src)
{
    //1 如果间隔 1s 到了，A 设备执行发送，数据变化
    if(send_flag == 1)
    {
        send_flag = 0;
        tx_msg.std_id = num & 0x7ff;
        tx_msg.ext_id = 0;//(u32)(num) & 0x3ffff;
        tx_msg.ide    = 0;//(u8 )(num) & 0x01;
        tx_msg.rtr    = 0;//(u8 )(num) & 0x01;
        tx_msg.dlc    = 8;
        tx_msg.data1  = num;
        tx_msg.data2  = num+1;
        can_send_msg(DEV0_BASEADDR,&tx_msg);
    }
    delay_ms(100);
    //2 如果 B 设备接收到，就回发
    if(int_src->rx_fnsh == 1)
    {
        int_src->rx_fnsh = 0;
        can_rec_msg(DEV1_BASEADDR,&rx_msg);
        rx_msg.std_id = (rx_msg.std_id + 0x400) & 0x7ff;
        can_send_msg(DEV1_BASEADDR,&rx_msg);
    }
    return 0;
}
```

接收的数据再次发送出去的时候，将 ID 加上 0x400 再发送，便于区分。完整代码以及其他相关的库函数请自行查看对应工程中的代码。

1.10板级验证

1.10.1实验所需硬件

（1）AC201-SA5Z-50D0 开发板

（2）FPGA 下载器：XIST USB Cable

（3）CM3 仿真器：DAP Link

（4）电源线

（5）Type-C 线

（6）CAN 模块 AC_CANFD_RS485

（7）CAN 调试器 BUSMUST

1.10.2硬件连接

本次实验的硬件连接如下图所示。

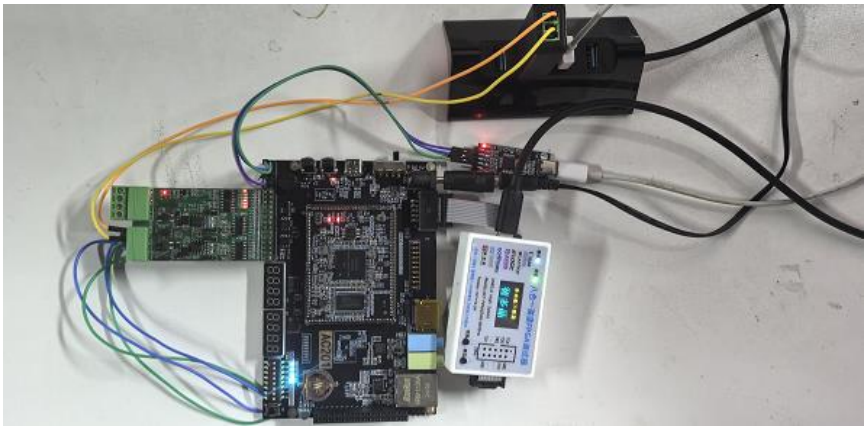


图 1-23 硬件连接

因为我们是 CAN 回环实验，我们需要将 CAN 模块 AC_CANFD_RS485 的 can0_H 和 can1_L 连接，can0_L 和 can1_H 连接。然后将 can 调试器 BUSMUST 的 H 端与 AC_CANFD_RS485 模块的 CAN0 或者 CAN1 的 H 端连接，L 端与 L 端连接。

1.10.3扫描 BUSMUST USB-CAN(FD)设备

我们打开 BUSMASTER 软件，扫描 BUSMUST USB-CAN(FD)设备，找到设备之后，设置波特率为 1M，如下所示。

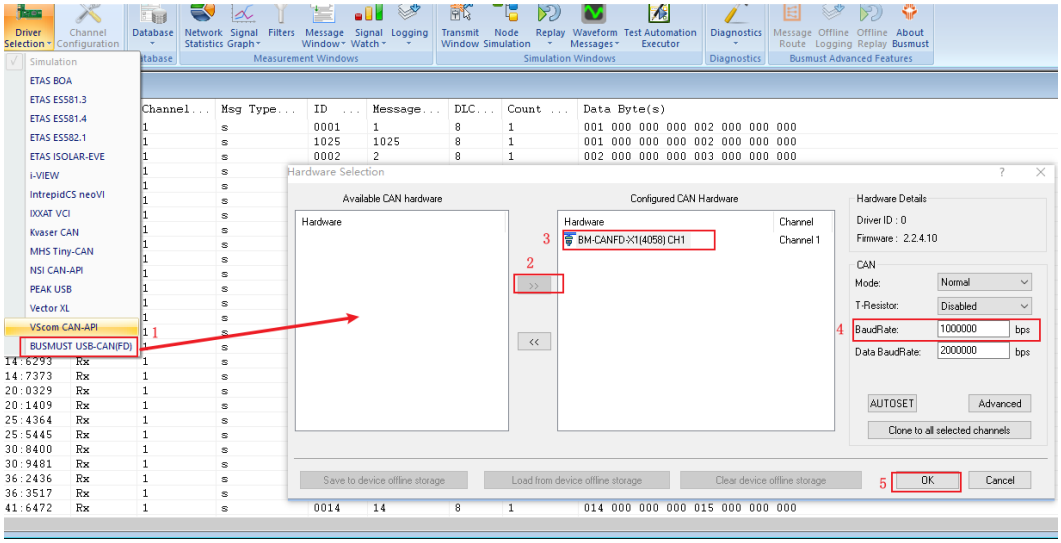


图 1-24 扫描设备

然后点击 Connect 进行设备连接。

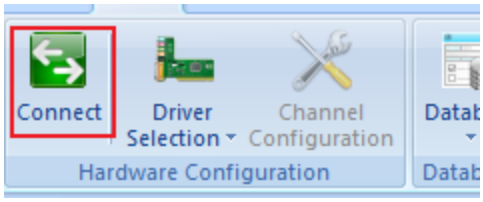


图 1-25 设备连接

1.10.4下载程序

1. 下载硬件设计文件，硬件测试文件在 HQ 工程目录下的 hq_run 文件夹下，具体操作方式如下所示。

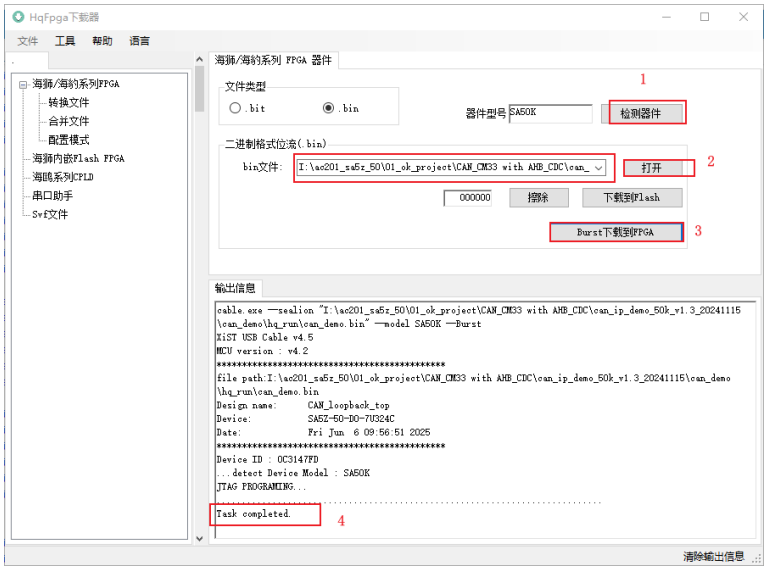


图 1-26 下载硬件设计文件

2. 下载软件设计文件，操作方式如下所示。

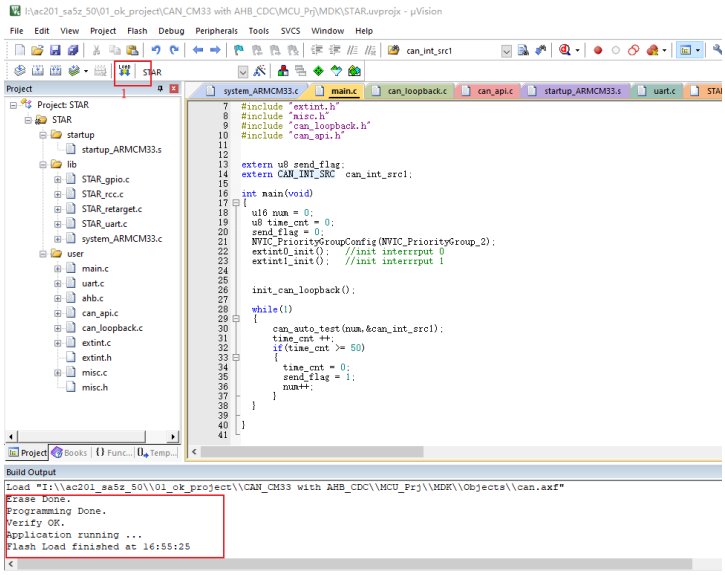
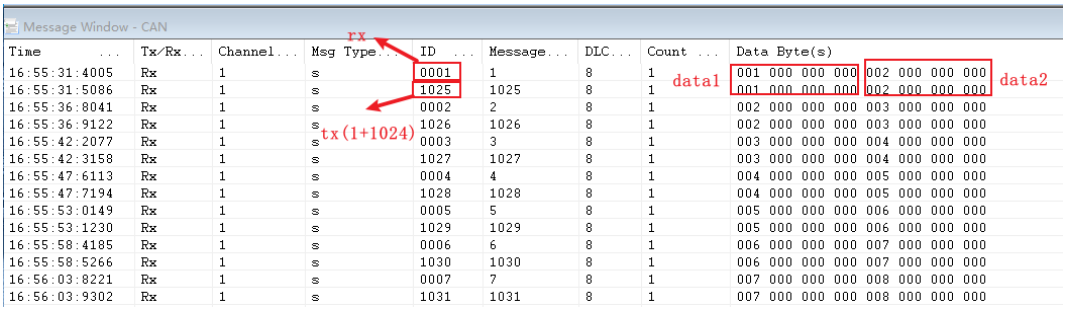


图 1-27 下载软件设计文件

1.10.5功能演示

此时我们就可以看到 BUSMASTER 软件接收到的数据如下所示。



Time	Tx/Rx	Channel	Msg Type	ID	Message	DLC	Count	Data Byte(s)
16:55:31:4005	Rx	1	s	0001	1	8	1	001 000 000 000 002 000 000 000
16:55:31:5086	Rx	1	s	1025	1025	8	1	001 000 000 000 002 000 000 000
16:55:36:8041	Rx	1	s	0002	2	8	1	002 000 000 000 003 000 000 000
16:55:36:9122	Rx	1	s	1026	1026	8	1	002 000 000 000 003 000 000 000
16:55:42:2077	Rx	1	s	0003	3	8	1	003 000 000 000 004 000 000 000
16:55:42:3158	Rx	1	s	1027	1027	8	1	003 000 000 000 004 000 000 000
16:55:47:6113	Rx	1	s	0004	4	8	1	004 000 000 000 005 000 000 000
16:55:47:7194	Rx	1	s	1028	1028	8	1	004 000 000 000 005 000 000 000
16:55:53:0149	Rx	1	s	0005	5	8	1	005 000 000 000 006 000 000 000
16:55:53:1230	Rx	1	s	1029	1029	8	1	005 000 000 000 006 000 000 000
16:55:58:4185	Rx	1	s	0006	6	8	1	006 000 000 000 007 000 000 000
16:55:58:5266	Rx	1	s	1030	1030	8	1	006 000 000 000 007 000 000 000
16:56:03:8221	Rx	1	s	0007	7	8	1	007 000 000 000 008 000 000 000
16:56:03:9302	Rx	1	s	1031	1031	8	1	007 000 000 000 008 000 000 000

图 1-28BUSMASTER 软件接收到的数据

从上图可以看出，第一行是代码中 DEV0 发送的数据，第二行为 DEV1 接收 DEV0 的数据然后再发送出去的，可以看到数据一致，并且 DEV1 的 ID 为 DEV0 的 ID 加上 1024，与代码中设置的一致，由此证明实验成功。