Application Note

使用 MSPM0 MCU 在 CAN 和 UART 之间进行桥接器设计

Yuhao Zhao

摘要

本应用手册介绍了 CAN 转 UART 桥接器。本文档描述了 CAN 转 UART 桥接器的结构和行为。本文档详细介绍了软件实现、硬件连接和应用程序用法。用户可以通过修改预定义来配置桥接器。此外，还提供了相关代码。

内容

插图清单

表格清单

商标

所有商标均为其各自所有者的财产。

1 引言

基于不同的应用，器件之间有多种通信方法。如今，MCU 通常支持一种以上的通信方法。例如，MSPM0 可以针对特定器件支持 UART、SPI、CAN 等。当器件通过不同的通信接口传输数据时，会构建一个桥。

对于 CAN 和 UART，CAN-UART 桥接器用作两个接口之间的转换器。CAN-UART 桥接器使器件能够在一个接口上发送和接收信息，并在另一个接口上接收和发送信息。

本应用手册介绍了用于创建和使用 CAN-UART 桥接器的软件和硬件设计。通过提供 CAN 和 UART 通信接口，MSPM0G3507 微控制器 (MCU) 可用作解决方案。随附的演示使用具有 2Mbps CANFD 和 9600 波特率 UART 的 MSPM0G3507 来演示通道之间的数据收发。

1.1 CAN 和 UART 之间的桥接器

CAN-UART 桥接器连接 CAN 和 UART 接口。本文中的示例可依靠 Launchpad 上的 XDS110 来使用 PC 观察 UART 数据。用户还可以通过 CAN-UART 桥接器从 PC 向 CAN 总线发送消息。对于 CAN 总线数据，用户可以使用 CAN 分析器或两个 LaunchPAD 形成环路，如随附演示的基本结构所示。

本文中的示例同时支持透明传输和协议传输。图 1-1 所示为用于透明传输的 PC 终端程序。图 1-2 所示为用于协议传输的 PC 终端程序。

对于协议传输，此示例指定了 UART 消息格式。用户还可以根据应用程序需要修改格式。接收来自 UART 的消息时，消息格式为 < 55 AA ID1 ID2 ID3 ID4 Length Data1 Data2 ...>。用户可以通过输入相同格式的数据，将数据从终端发送到 CAN 总线。55 AA 是标头。ID 区域为四字节。长度区域为一字节，表示数据长度。

对于透明传输，UART 使用可配置的超时来检测一条消息。来自 UART 的数据被填充到 CAN 的数据区域（反向相同）。CAN ID 是默认值。

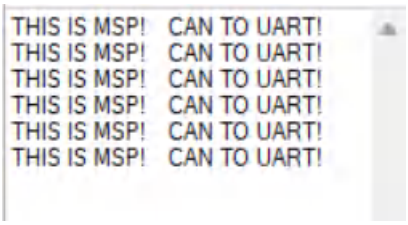

图 1-1. 用于透明传输的 PC 终端程序

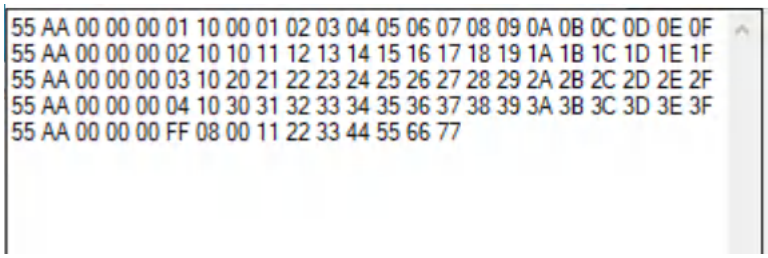

图 1-2. 用于协议传输的 PC 终端程序

2 实施

2.1 原理

在本文的设计中，CAN-UART 桥接器同时使用 CAN 接收和发送以及 UART 接收和发送。因此 CAN 模块和 UART 模块都必须进行配置。由于不同通信的消息格式不同，CAN-UART 桥接器还必须转换消息格式。

对于 CAN，CAN 模块支持传统 CAN 和 CAN FD（具有灵活数据速率的 CAN）协议。CAN 模块符合 ISO 11898-1:2015 标准。如需更多信息，请参阅相关文档。对于 UART，该接口可用于在 MSPM0 器件和另一个采用串行异步通信协议的器件之间传输数据。如需更多信息，请参阅相关文档。

图 2-1 展示了 CAN-UART 桥接器的基本原理。通常，CAN 的通信速率远高于 UART 的通信速率。对于 CAN FD，波特率可高达 5Mbps，而 UART 以 9600bps 的波特率工作，如示例代码所示。因此，CAN 接收到的数据可能无法及时由 UART 发送。为了匹配波特率，该方案使用缓冲器在 CAN 和 UART 之间传输数据。此缓冲器不仅实现了数据缓存，还实现了数据格式转换。这相当于在两个通信接口之间添加屏障。用户可以为过载情况添加过载控制操作。

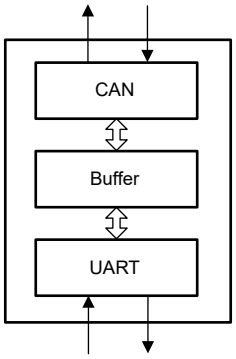

图 2-1. CAN-UART 桥接器的基本原理

2.2 结构

可以在 图 2-3 中看到具有协议传输功能的 CAN-UART 桥接器的结构。CAN-UART 桥接器可以分为四个独立的任务：从 UART 接收、从 CAN 接收、通过 CAN 传输、通过 UART 传输。两个 FIFO 实现双向消息传输和消息缓存。

图 2-4 展示了具有透明传输功能的 CAN-UART 桥接器的结构。添加了计时器中断，用于在一个数据包结束时检测超时。

UART 和 CAN 接收均设置为中断触发，以便能及时接收消息。进入中断时，首先通过 getXXXRxMsg() 提取消息。

对于 CAN，CAN 帧是固定格式。MSPM0 支持传统 CAN 或 CANFD。CANFD 的帧如 图 2-2 所示。本文中的示例可以在数据区域中为协议传输定义零、一和四个字节的附加 ID。

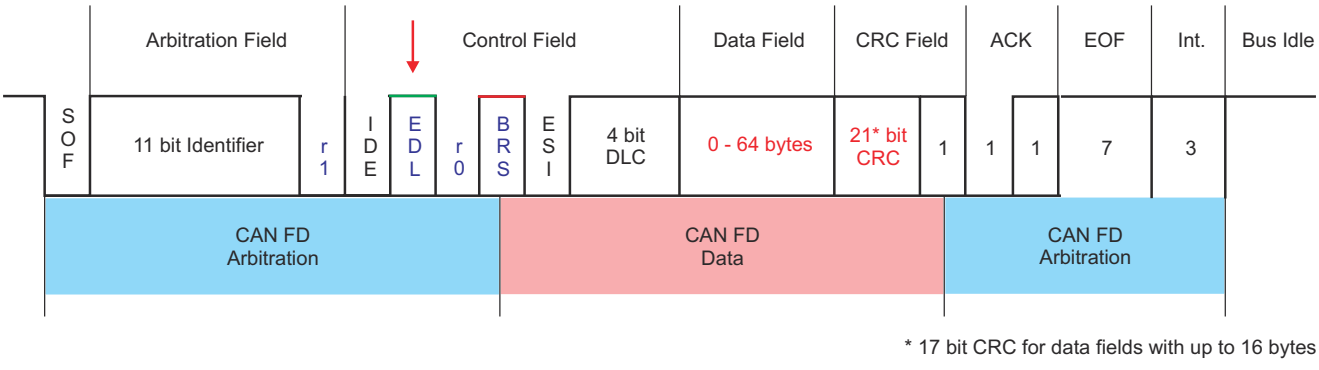

图 2-2. CAN FD 帧

表 2-1. CAN 数据包格式

	ID 区域	数据
协议传输	4/1/0 字节	(数据长度) 字节

对于 UART 协议传输，根据串行帧信息来识别消息。UART 消息格式在 UART 数据包格式中列出。

表 2-2. UART 数据包格式

	报头	ID 区域	数据长度	数据
协议传输	0×55 0×AA	4/1/0 字节	1 字节	(数据长度) 字节
透明传输	—	—	—	(数据长度) 字节

报头是一个固定的十六进制数字，组合为 *0x55 0xAA*，表示组的开头。ID 区域默认占用四个字节以匹配 CAN ID，可配置为一个字节，或者 ID 区域不存在。数据长度区域占用一个字节。数据长度区域之后是特定长度的数据。此格式作为示例提供。用户可以根据应用的要求修改格式。

对于 UART 透明传输，发生超时时会识别消息。所有字节都被视为纯数据。默认值是数据包信息的装载。(例如 ID)。

收到消息后，processXXXRxMsg() 转换消息的格式，并将消息作为新元素存储在 FIFO 中。FIFO 元素的格式如 图 2-5 所示。在 FIFO 元素的格式中，FIFO 元素中有四个类别：*来源 ID*、*目标 ID*、*数据长度* 和 *数据*。用户还可以根据应用的要求更改消息项目。此外，该方案还会检查 FIFO 是否已满以进行过载控制。用户可以根据应用要求添加过载控制操作。

CAN 和 UART 传输均在主函数中执行。当检测到 FIFO 非空时，将提取 FIFO 元素。消息将被格式化并发送。对于 CAN，CAN 帧是固定格式。对于 UART，消息以 CAN 数据包格式中所述的格式发送。在本文的设计中，使用 UART TX 中断将数据填充到 UART TX 缓冲区中。

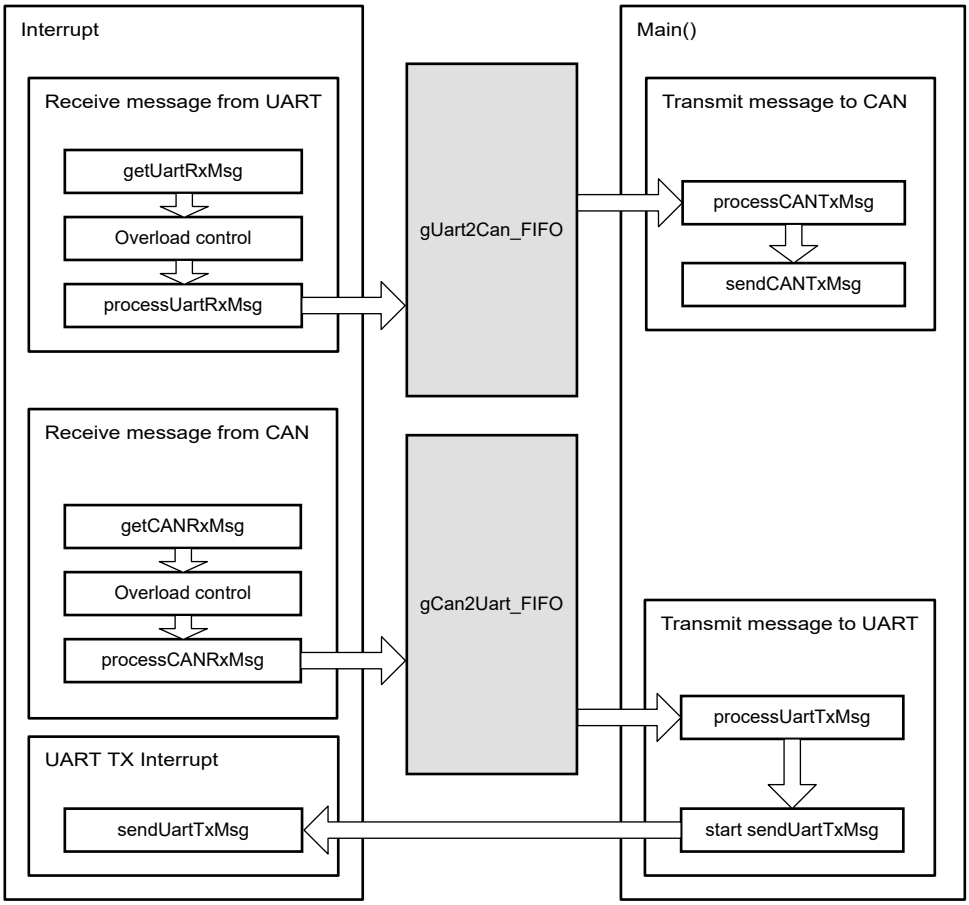

图 2-3. CAN-UART 桥接器的结构：协议

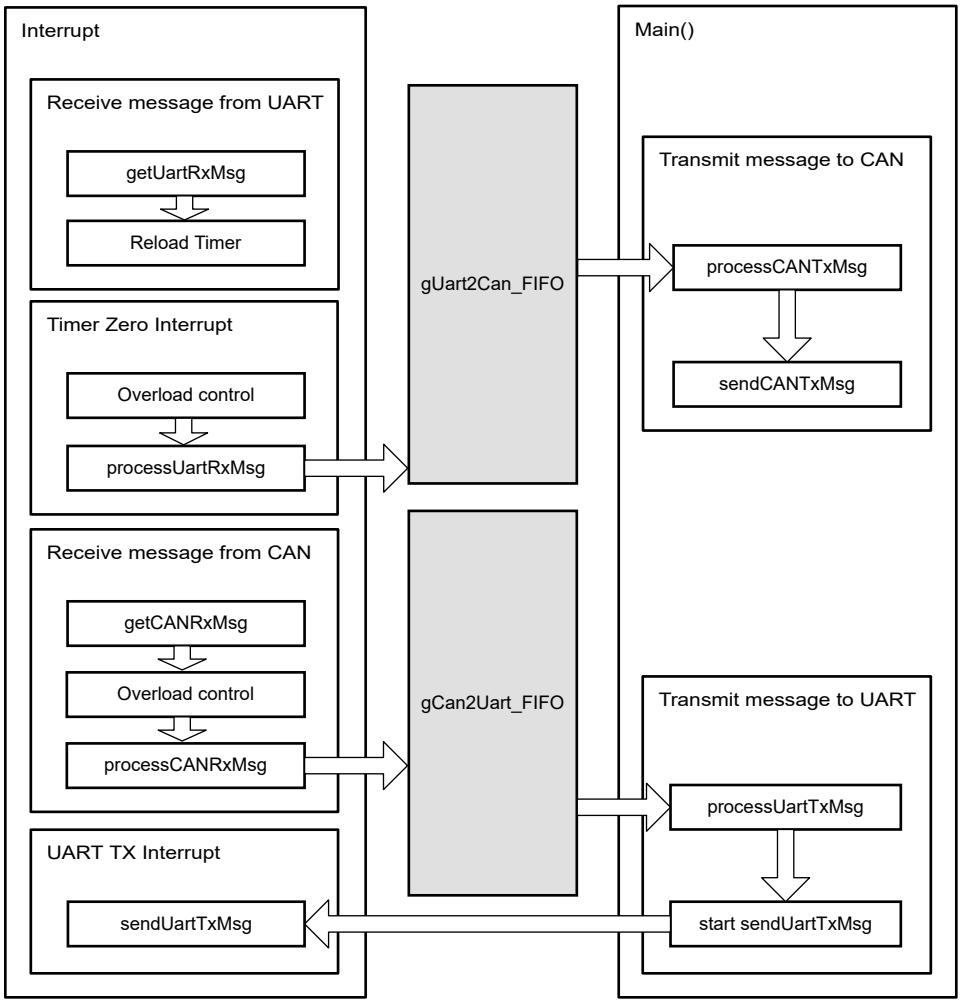

图 2-4. CAN-UART 桥接器的结构：透明

FIFO 结构如 图 2-5 所示。每个 FIFO 使用三个全局变量来指示 FIFO 状态。*gUart2Can_FIFO, gUart2Can_FIFO.fifo_in* 指示写入位置，*gUart2Can_FIFO.fifo_out* 指示读取位置，gUart2Can_FIFO.fifo_Count 指示 *gUart2Can_FIFO* 中的元素数量。

如果 *gUart2Can_FIFO* 为空，*gUart2Can_FIFO.fifo_in* 等于 *gUart2Can_FIFO.fifo_out*，并且 *gUart2Can_FIFO.fifo_count* 为零。

执行 *processUartRxMsg()* 时，来自 UART 的新消息将存储到 *gUart2Can_FIFO* 中。因此，*gUart2Can_FIFO.fifo_in* 会移动到下一个位置，并且 *gUart2Can_FIFO.fifo_count* 会递增 1。

将消息从 *gUart2Can_FIFO* 传输到 CAN 时，*gUart2Can_FIFO.fifo_out* 会移动到下一个位置，而 *gUart2Can_FIFO.fifo_count* 会减去 1。*gCan2Uart_FIFO* 与 *gUart2Can_FIFO* 类似。

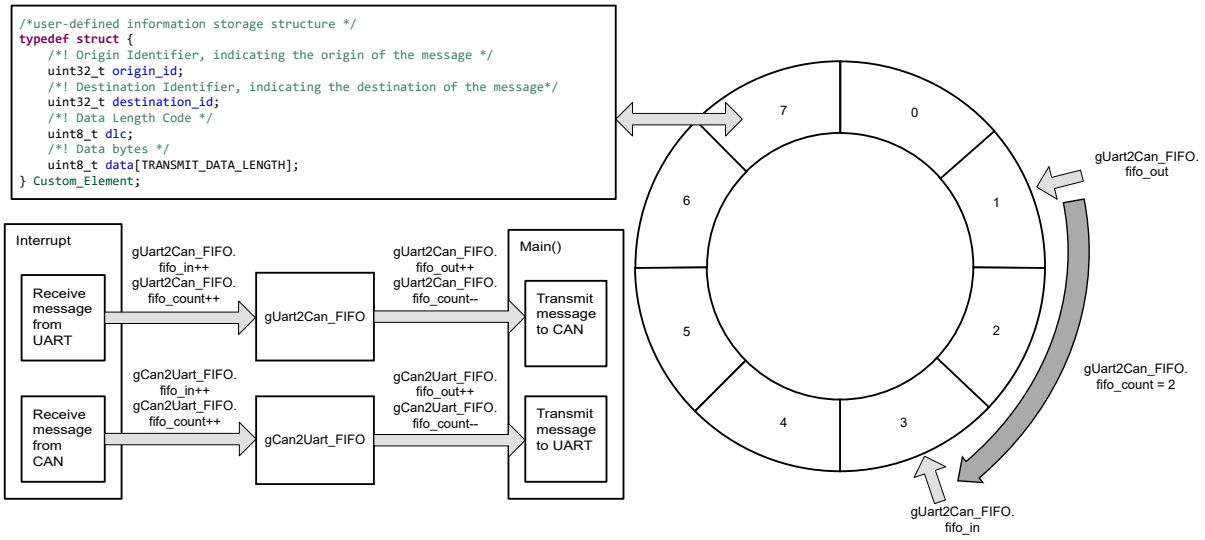

图 2-5. FIFO 的结构

3 软件说明

3.1 软件功能

图 2-3 展示了功能设计。表 3-1 列出了函数。

表 3-1. 函数和说明

任务	函数	说明	位置
UART 接收	getUartRxMsg()	获取接收到的 UART 消息	bridge_uart.c bridge_uart.h
	processUartRxMsg()	转换接收到的 UART 消息格式，并将消息存储到 gUART_RX_Element 中	
UART 发送	processUartTxMsg()	转换要通过 UART 发送的 gUART_TX_Element 格式	
	sendUartTxMsg()	通过 UART 发送消息	
CAN 接收	getCANRxMsg()	获取接收到的 CAN 消息	bridge_can.c bridge_can.h
	processCANRxMsg()	转换接收到的 CAN 消息格式，并将消息存储到 gCAN_RX_Element 中	
CAN 发送	processCANTxMsg()	转换要通过 CAN 发送的 gCAN_TX_Element 格式	
	sendCANTxMsg()	通过 CAN 发送消息	

3.2 可配置参数

所有可配置参数都在 user_define.h 中定义，这些参数在可配置参数中列出。

对于 UART，此示例同时支持透明传输和协议传输。可以通过定义 UART_TRANSPARENT 或 UART_PROTOCOL 来切换这些函数。

在透明传输中，用户可以配置超时来检测一条 UART 消息接收是否完成。

在协议传输中，用户可以为不同格式配置 ID 长度。请注意，有一个固定的 2 字节报头 (0×55 0×AA) 和 1 字节数据长度。若要更详细地修改格式，用户可能需要直接修改代码。

```
#define UART_TRANSPARENT
#ifdef UART_TRANSPARENT
/* The format of Uart:
     * Transparent transmission - Data1 Data2 ...*/
#define UART_TIMEOUT  (0x4000)      //timeout 250ms
#else
#define UART_PROTOCOL
/* The format of Uart:
     * if UART_ID_LENGTH = 4, format is 55 AA ID1 ID2 ID3 ID4 Length Data1 Data2 ...
     * if UART_ID_LENGTH = 1, format is 55 AA ID Length Data1 Data2 ...
     * if UART_ID_LENGTH = 0, format is 55 AA Length Data1 Data2 ...*/
//#define UART_ID_LENGTH  (0)
//#define UART_ID_LENGTH  (1)
#define UART_ID_LENGTH  (4)
#endif
```

对于 CAN，ID 和数据长度包含在 CAN 帧中。用户可以通过更改 CAN_ID_LENGTH 在数据区域中添加另一个 ID。（默认值为 0）。

```
/* The format of CAN:
     * if CAN_ID_LENGTH = 4, format is ID1 ID2 ID3 ID4 Data1 Data2 ...
     * if CAN_ID_LENGTH = 1, format is ID Data1 Data2 ...
     * if CAN_ID_LENGTH = 0, format is Data1 Data2 ...*/
#define CAN_ID_LENGTH  (0)
//#define CAN_ID_LENGTH  (1)
//#define CAN_ID_LENGTH  (4)
```

表 3-2. 可配置参数

参数	可选值	说明
#define UART_TRANSPARENT	定义/未定义	启用 UART 透明传输。
#define UART_PROTOCOL	定义/未定义	启用 UART 协议传输。
#define UART_TIMEOUT (0x4000)	超时 = UART_TIMEOUT/ 32768s	超时表示一条 UART 消息接收完成。仅在定义了 UART_TRANSPARENT 时可用。在本例中，默认值为 250ms。
#define UART_ID_LENGTH (4)	0/1/4	可选 UART ID 长度，与协议中的 ID 区域相关。仅在定义了 UART_PROTOCOL 时可用。在本例中，默认值为四字节。
#define CAN_ID_LENGTH (0)	0/1/4	可选 CAN ID 长度，与协议中的 ID 区域相关。在本例中，默认值为 0 字节。
#define TRANSMIT_DATA_LENGTH (12)	<=64	数据区域的大小。如果接收到的消息包含的数据多于此值，可能会发生数据丢失。
#define C2U_FIFO_SIZE (8)		CAN 到 Uart 的 FIFO 大小。请注意 SRAM 的用法。
#define U2C_FIFO_SIZE (8)		Uart 到 CAN 的 FIFO 大小。请注意 SRAM 的用法。
#define DEFAULT_UART_ORIGIN_ID (0x00)		UART 原始 ID 的默认值
#define DEFAULT_UART_DESTINATION_ID (0x00)		UART 目标 ID 的默认值
#define DEFAULT_CAN_ORIGIN_ID (0x00)		CAN 原始 ID 的默认值
#define DEFAULT_CAN_DESTINATION_ID (0x00)		CAN 目标 ID 的默认值

3.3 定制元素结构

Custom_Element 结构在 *user_define.h* 中定义。*Custom_Element* 也显示在 图 2-5 中。

来源标识符指示消息的来源。以下是示例(CAN_ID_LENGTH =0,UART_ID_LENGTH =4)。

- 示例 1 - CAN 接口接收和传输
  - 当 CAN-UART 桥接器接收到 CAN 消息时，CAN 帧的 ID 是*来源标识符*，其指示消息的来源。
  - 当 CAN-UART 桥接器传输 CAN 消息时，将忽略来源标识符（CAN_ID_LENGTH 默认设置为 0）。
- 示例 2 - UART 接口接收和传输（UART 协议传输）
  - 当 CAN-UART 桥接器接收到 UART 消息（UART 协议传输）时，由于 UART 没有 ID，*DEFAULT_UART_ORIGIN_ID* 是*来源标识符*。
  - 当 CAN-UART 桥接器传输 UART 消息（UART 协议传输）时，UART 数据中的*来源标识符*将是 4 字节 ID（UART_ID_LENGTH 默认设置为 4），其指示消息来源。
- 示例 3 - UART 接口接收和传输（UART 透明传输）
  - 当 CAN-UART 桥接器接收到 UART 消息（UART 透明传输）时，由于 UART 没有 ID，*DEFAULT_UART_ORIGIN_ID* 是*来源标识符*。
  - 当 CAN-UART 桥接器传输 UART 消息（UART 透明传输）时，将忽略*来源标识符*（透明传输没有 ID 区域）。

*目标标识符*指示消息的目标。以下是示例(CAN_ID_LENGTH =0,UART_ID_LENGTH =4)。

- 示例 1 - CAN 接口接收和传输
  - 当 CAN-UART 桥接器接收 CAN 消息时，由于 CAN_ID_LENGTH 默认设置为 0，DEFAULT_CAN_DESTINATION_ID 是*目标标识符*。UART 传输不需要 ID。
  - 当 CAN-UART 桥接器传输 CAN 消息时，*目标标识符*将是 CAN 帧中的 CAN ID。在此示例中，11 位和 29 位均受支持。
- 示例 2 - UART 接口接收和传输（UART 协议传输）
  - 当 CAN-UART 桥接器接收到 UART 消息（UART 协议传输）时，来自 UART 数据的 4 字节 ID 是*目标标识符*（UART_ID_LENGTH 默认设置为 4）。CAN 传输需要 ID 信息。
  - 当 CAN-UART 桥接器传输 UART 消息（UART 协议传输）时，由于 UART 传输不需要 ID，将忽略*目标标识符*。
- 示例 3 - UART 接口接收和传输（UART 透明传输）
  - 当 CAN-UART 桥接器接收 UART 消息（UART 透明传输）时，DEFAULT_UART_DESTINATION_ID 是*目标标识符*。（透明传输没有 ID 区域）。CAN 传输需要 ID 信息。
  - 当 CAN-UART 桥接器传输 UART 消息（UART 透明传输）时，由于 UART 传输不需要 ID，将忽略*目标标识符*。

```
/*user-defined information storage structure */
typedef struct {
    /*! Origin Identifier, indicating the origin of the message */
    uint32_t origin_id;
    /*! Destination Identifier, indicating the destination of the message */
    uint32_t destination_id;
    /*! Data Length Code */
    uint8_t dlc;
    /*! Data bytes */
    uint8_t data[TRANSMIT_DATA_LENGTH];
} Custom_Element;
```

3.4 FIFO 的结构

Custom_FIFO 结构在 *user_define.h* 中定义。这也显示在 图 2-5 中。

```
typedef struct {
    uint16_t fifo_in;
    uint16_t fifo_out;
    uint16_t fifo_count;
```

```
    Custom_Element *fifo_pointer;
} Custom_FIFO;
```

gCan2Uart_FIFO 和 gUart2Can_FIFO 在 main.c 中定义请注意 SRAM 的使用情况，该使用情况与 C2U_FIFO_SIZE、U2C_FIFO_SIZE 和 Custom_Element 的大小相关。

```
/* Variables for C2U_FIFO
 * C2U_FIFO is used to temporarily store message from CAN to UART */
Custom_Element gC2U_FIFO[C2U_FIFO_SIZE];
Custom_FIFO gCan2Uart_FIFO = {0, 0, 0, gC2U_FIFO};

/* Variables for U2C_FIFO
 * U2C_FIFO is used to temporarily store message from UART to CAN */
Custom_Element gU2C_FIFO[U2C_FIFO_SIZE];
Custom_FIFO gUart2Can_FIFO = {0, 0, 0, gU2C_FIFO};
```

3.5 UART 接收和传输（透明传输）

对于 *UART 接收*，在 *bridge_uart.c* 中定义了三个全局变量。

```
uint8_t gUartReceiveGroup[UART_RX_SIZE];
Custom_Element gUART_RX_Element;
uint16_t gGetUartRxMsg_Count;
```

下面介绍 *UART 接收*的过程

1. 调用 *getUartRxMsg_transparent()* 以将消息存储到 *gUartReceiveGroup* 中。当发生超时或组已满时（数据高达 *TRANSMIT_DATA_LENGTH* 字节），完成消息接收
2. 调用 *processUartRxMsg_transparent()* 以从 *gUartReceiveGroup* 中提取数据，并将数据存储到 *gUART_RX_Element* 中。
3. 将 *gUART_RX_Element* 放入 *gUart2Can_FIFO*。

对于 UART 传输，在 bridge_uart.c 中定义了两个全局变量。

```
uint8_t gUartTransmitGroup[UART_TX_SIZE];
Custom_Element gUART_TX_Element;
```

下面介绍 *UART 传输*的过程。

1. 从 *gCan2Uart_FIFO* 接收 *gUART_TX_Element*。
2. 调用 *processUartTxMsg_transparent()* 以从 *gUART_TX_Element* 获取数据，并将其存储到 *gUartTransmitGroup* 中。
3. 调用 *sendUartTxMsg()* 以通过 UART 传输 *gUartTransmitGroup*。

3.6 UART 接收和传输（协议传输）

对于 *UART 接收*，在 *bridge_uart.c* 中定义了两个全局变量。

```
uint8_t gUartReceiveGroup[UART_RX_SIZE];
Custom_Element gUART_RX_Element;
```

下面介绍 UART 接收的过程。

1. 调用 *getUartRxMsg()* 以检测报头，将完整的消息存储到 *gUartReceiveGroup* 中。
2. 调用 *processUartRxMsg()* 以从 *gUartReceiveGroup* 中提取信息并将信息存储在 *gUART_RX_Element* 中。
3. 将 *gUART_RX_Element* 放入 *gUart2Can_FIFO*。

对于 *UART 传输*，在 *bridge_uart.c* 中定义了两个全局变量。

```
uint8_t gUartTransmitGroup[UART_TX_SIZE];
Custom_Element gUART_TX_Element;
```

下面介绍 UART 传输的过程。

1. 从 *gCan2Uart_FIFO* 获取 *gUART_TX_Element*。
2. 调用 *processUartTxMsg()* 以从 *gUART_TX_Element* 获取信息，并将信息存储到 *gUartTransmitGroup* 中。
3. 调用 *sendUartTxMsg()* 以通过 UART 传输 *gUartTransmitGroup*。

3.7 CAN 接收和传输

对于 *CAN 接收*，在 *bridge_can.c* 中定义了两个全局变量。

```
DL_MCAN_RxBufElement rxMsg;
Custom_Element gCAN_RX_Element;
```

下面介绍 *CAN 接收* 的过程。

1. 调用 *getCANRxMsg()* 以获取从 *CAN 消息 RAM* 到 *rxMsg* 的完整消息。
2. 调用 *processCANRxMsg()* 从 *rxMsg* 中提取信息并将其存储到 *gCAN_RX_Element* 中。
3. 将 *gCAN_RX_Element* 放入 *gCan2Uart_FIFO* 中。

对于 *CAN 传输*，在 *bridge_can.c* 中定义了两个全局变量。

```
DL_MCAN_TxBufElement txMsg0;
Custom_Element gCAN_TX_Element;
```

下面介绍 CAN 传输的过程。

1. 从 *gUart2Can_FIFO* 获取 *gCAN_TX_Element*。
2. 调用 *processCANTxMsg()* 以从 *gCAN_TX_Element* 中提取信息并将其存储到 *txMsg0* 中。
3. 调用 *sendCANTxMsg ()* 以通过 CAN 传输 *txMsg0*。

3.8 应用集成

表 3-1 中的函数被分类到不同的文件中。UART 接收和传输函数包含在 *bridge_uart.c* 和 *bridge_uart.h* 中。CAN 接收和传输函数包含在 *bridge_can.c* 和 *bridge_can.h* 中。FIFO 元素结构在 *user_define.h* 中定义。

用户可以通过文件轻松分离函数。例如，如果只需要 UART 函数，用户可以保留 *bridge_uart.c* 和 *bridge_uart.h* 以调用相应函数。

对于外设的基本配置，该项目集成了 SysConfig 配置文件。用户可以使用 SysConfig 轻松修改外设的基本配置。

需要此功能的应用程序必须包含 CAN 模块 API 和 UART 模块 API。所有 API 文件都包含在下载的 SDK 中。

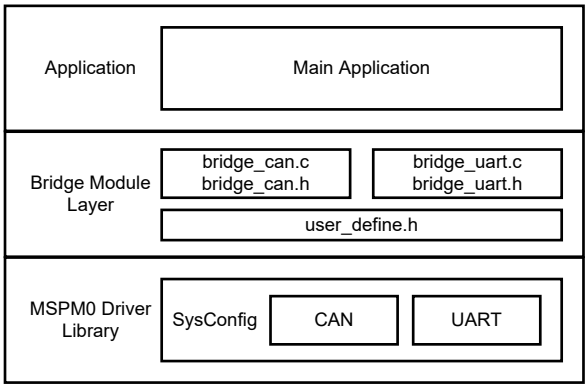

图 3-1. 软件所需的文件

表 3-3 列出了 CAN-UART 桥接器设计在闪存大小和 RAM 大小方面的占用空间。图 3-1 和 表 3-3 的制作是使用 Code Composer Studio（版本：12.7.1.00001）且优化级别为 2 的条件下确定的。

用户可以调整 FIFO 的大小。FIFO 越大，意味着缓存容量越大，但占用的 RAM 空间也越大。有关详细信息，请参阅 节 5 中的相关内容。此外，此代码中数据字段的大小默认设置为最大 64 字节。用户可以根据实际数据长度配置数据字段大小。使用 12 字节的数据字段能够显著减少 RAM 的使用，如 表 3-3 中所列

表 3-3. CAN-UART 桥接器的内存占用

所需的最小代码大小（字节）	闪存	SRAM
CAN-UART 桥接器 （协议传输） U2C_FIFO_SIZE=8 C2U_FIFO_SIZE = 8 数据大小 = 12 字节）	6328	910
CAN-UART 桥接器 （协议传输） U2C_FIFO_SIZE=8 C2U_FIFO_SIZE=8 数据大小 = 64 字节）	6416	2054
CAN-UART 桥接器 （协议传输 U2C_FIFO_SIZE=30 C2U_FIFO_SIZE=30 数据大小 = 12 字节）	6432	1966

4 硬件

通过 LaunchPad 上的 XDS110，用户可以使用 PC 在 UART 端发送和接收消息。作为演示，可以将两个 LaunchPad 用作两个 CAN-UART 桥接器以形成一个环路。当 PC 通过其中一个 LaunchPad 发送 UART 消息时，PC 将从另一个 LaunchPad™ 接收 UART 消息。图 4-1 展示了基本结构。请注意，构建 CAN 总线需要 CAN 收发器。

随附的演示使用两个 LaunchPad：一个 TCAN1046EVM 和一个 PC。TCAN1046EVM 是一款高速双通道 CAN 收发器评估模块。图 4-2 显示了演示的连接。对于 LaunchPad，PA12 用于 CAN 发送，PA13 用于 CAN 接收。PA12 和 PA13 应连接到 TCAN1046EVM 的 TX 引脚和 RX 引脚。PA20 用于 UART 发送，PA21 用于 UART 接收。请注意，LaunchPad 的 eZ-FET 上的反向通道 UART 接口可用于与 PC 的 UART 通信。

对于 TCAN1046EVM，VCC 必须连接到 5V，VIO 必须连接到 3.3V，因为 TCAN1046 支持电平移位。要构建 CAN 总线，CANH1 和 CANL1 必须连接到 CANH2 和 CANL2。此外，CAN 总线上的终端（CANH 和 CANL）必须配置 J2（或 J3）和 J6（或 J8）跳线。每个跳线都将 120Ω 终端添加到各自的总线。有关更多信息，请参阅相关文档。

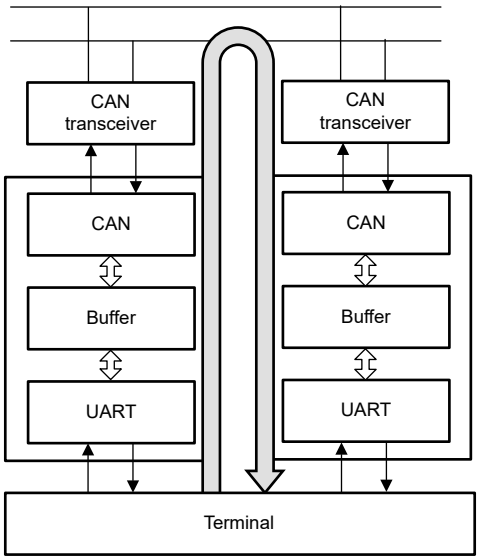

图 4-1. 随附演示的基本结构

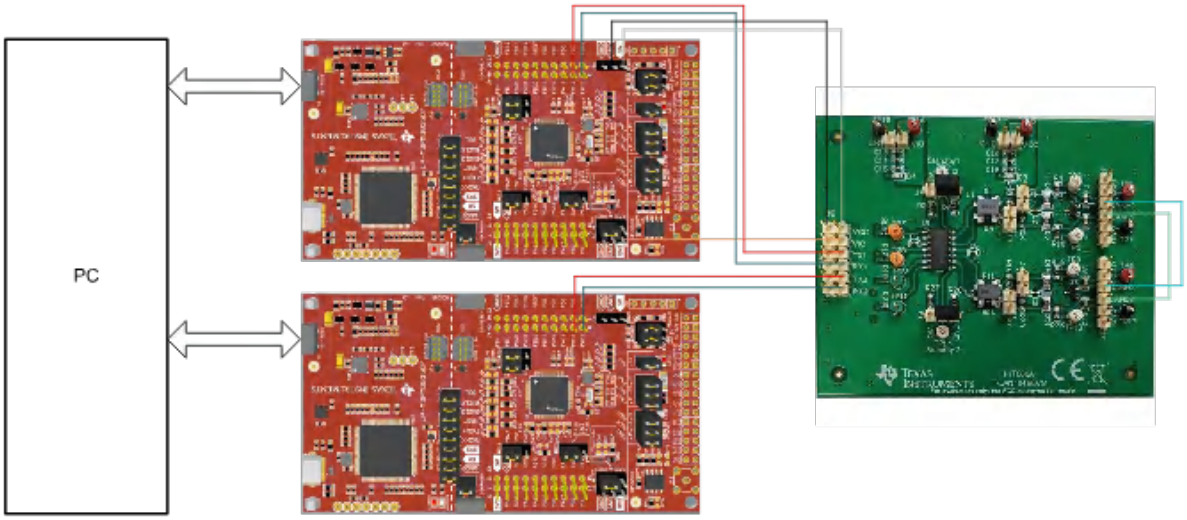

图 4-2. 演示的硬件连接

5 应用程序方面

本节介绍了 CAN-UART 桥接器设计在应用程序方面的特性，以及如何对设计进行配置以满足应用程序需求。

5.1 灵活的结构

有各种可配置参数，如 节 3.2 中所示。用户可以通过修改 *user_define.h* 中定义的所有参数来配置 CAN 和 UART 数据包帧、FIFO 的大小或数据区域的最大大小。

用户可以在 *user_define.h* 中修改 *Custom_Element* 的定义。可根据应用程序和存储要求增加或减少条目。

```
/*user-defined information storage structure */
typedef struct {
    /*! Origin Identifier, indicating the origin of the message */
    uint32_t origin_id;
    /*! Destination Identifier, indicating the destination of the message */
    uint32_t destination_id;
    /*! Data Length Code */
    uint8_t dlc;
    /*! Data bytes */
    uint8_t data[TRANSMIT_DATA_LENGTH];
} Custom_Element;
```

两个通信接口的接收和传输单独运行。消息通过 FIFO 传输。用户可以更改结构（例如使消息遵循特定的格式，甚至是特定的通信协议）。此外，用户可以根据 图 2-3 将结构拆分为单向传输。

5.2 CAN 的可选配置

MSPM0 的 CAN 模块符合 CAN 协议 2.0 A、B 和 ISO 11898-1:2015 标准。用户可以配置 CAN 模块的各种功能。通过使用 SysConfig，用户可以更改 CAN 的基本配置。（例如，数据传输速率）。

代码提供了 CAN ID 的可选配置。示例代码默认为 11 位 ID（标准 ID）。可以通过修改 *user_define.h* 来更改配置。

- 添加 *#define CAN_ID_EXTEND* 以启用 29 位 ID（扩展 ID）。

此外，该示例代码支持在单个帧中携带 64 字节的数据。用户可以根据需要配置适当的数据大小，从而进一步减少 FIFO 占用的 RAM 空间。

5.3 CAN 总线多节点通信示例

CAN 通信为总线通信。用户可以使用 CAN-UART 桥接器设计来测试 CAN 总线的多节点通信。图 5-1 显示了基本结构。当用户通过任何 CAN-UART 桥接器向 CAN 总线发送消息时，系统会立即从其他节点读回该消息。

必须使用至少三个 LaunchPad。LaunchPad 上的每个 CAN 通信都需要一个收发器。LaunchPad 和收发器之间的连接如 图 4-2 所示。

MSPM0 的 CAN 模块支持硬件滤波，以选择具有特定 ID 的消息。请注意，在此示例代码中，默认情况下不执行硬件滤波。用户可以配置硬件滤波。有关具体配置，请参阅相关文档。

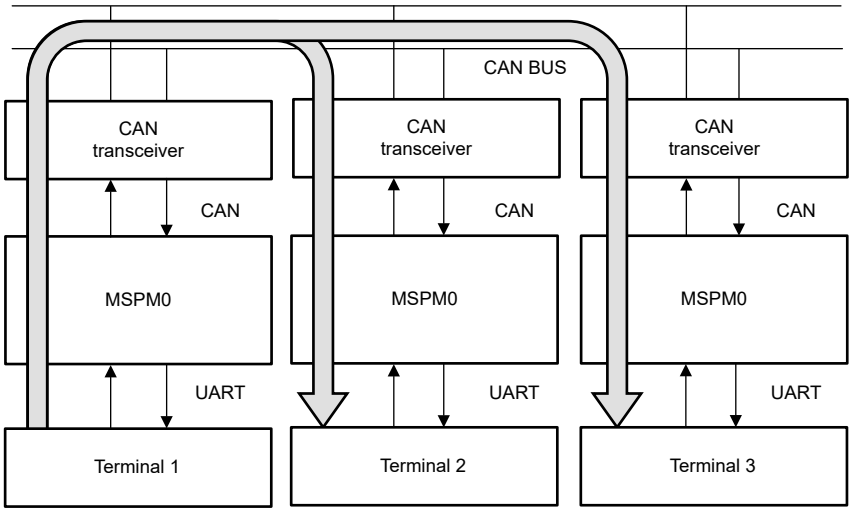

图 5-1. 多节点通信的基本结构

6 总结

本文档介绍了 CAN 转 UART 桥接器的实现，包括结构、函数定义、接口使用和应用方面。通过此示例，MSPM0 可以充当 CAN 和 UART 之间的转换器，允许用户在一个接口上发送和接收信息，在另一个接口上接收和发送信息。

7 参考资料

- 德州仪器 (TI)，*使用 MSPM0 MCU 在 CAN 和 SPI 之间建立桥接器解决方案* 应用手册。
- 德州仪器 (TI)，*使用 MSPM0 MCU 在 CAN 和 I2C 之间建立桥接器解决方案* 应用手册。
- 德州仪器 (TI)，*CAN 转 UART 桥接器* 子系统设计。
- 德州仪器 (TI)，CAN 转 SPI 桥接器 子系统设计。
- 德州仪器 (TI)，CAN 转 I2C 桥接器 子系统设计。
- 德州仪器 (TI)，下载 MSPM0 SDK 资源管理器。
- 德州仪器 (TI)，详细了解 SysConfig 系统配置工具。

重要通知和免责声明

TI“按原样”提供技术和可靠性数据（包括数据表）、设计资源（包括参考设计）、应用或其他设计建议、网络工具、安全信息和其他资源，不保证没有瑕疵且不做出任何明示或暗示的担保，包括但不限于对适销性、某特定用途方面的适用性或不侵犯任何第三方知识产权的暗示担保。

这些资源可供使用 TI 产品进行设计的熟练开发人员使用。您将自行承担以下全部责任：(1) 针对您的应用选择合适的 TI 产品，(2) 设计、验证并测试您的应用，(3) 确保您的应用满足相应标准以及任何其他功能安全、信息安全、监管或其他要求。

这些资源如有变更，恕不另行通知。TI 授权您仅可将这些资源用于研发本资源所述的 TI 产品的相关应用。 严禁以其他方式对这些资源进行复制或展示。您无权使用任何其他 TI 知识产权或任何第三方知识产权。您应全额赔偿因在这些资源的使用中对 TI 及其代表造成的任何索赔、损害、成本、损失和债务，TI 对此概不负责。

TI 提供的产品受 TI 的销售条款或 ti.com 上其他适用条款/TI 产品随附的其他适用条款的约束。TI 提供这些资源并不会扩展或以其他方式更改 TI 针对 TI 产品发布的适用的担保或担保免责声明。

TI 反对并拒绝您可能提出的任何其他或不同的条款。

邮寄地址：Texas Instruments, Post Office Box 655303, Dallas, Texas 75265