茂晶有限公司 GFE International (Hong Kong) Limited A MACNICA Company

XMOS 特許一級代理商

茂晶有限公司
國內服務據點 ：
深圳：Tel: 86-755-8828 5788-1207 Email: Shenzhen@gfei.com.hk
北京：Tel: 86-10-5126 6624 Email: Beijing@gfei.com.hk
上海：Tel: 86-21-54453155 Email: Shanghai@gfei.com.hk
武漢：Tel: 86-27-8730 6822, 8784 0783 Email: Wuhan@gfei.com.hk
青島：Tel: 86-532-8573 1420 Email: Qingdao@gfei.com.hk
成都：Tel: 86-028-85548390 Email: Chengdu@gfei.com.hk
廈門：Tel: 86-0592-5302668 Email: Xiamen@gfei.com.hk
海外服務據點 ：
香港：Tel: 852-3741 0662-2293 Email: Hongkong@gfei.com.hk
台灣：Tel: 886-2-89132200 Email: Service@gfei.com.hk
技術支援 ：
深圳：Tel: 86-755-8828 5788-1207 Email: royl@gfei.com.hk
上海：Tel: 86-21-54453155
北京：Tel: 86-10-8429 8668 Email: jackl@gfei.com.hk

CAN to Ethernet bridge

自从完整的通信协议在多核处理器的软件环境里得到实现之后，许多灵活的桥系统能被建立为了实现你的精确的需求。
这个应用笔记展示了如何创造一个单路 CAN 协议转换成以太网桥，这个应用桥功能的实现是为了展示如何从 CAN 网络的 CAN 框架转换成以太网（TCP 包）等，经由 XMOS 设备以太网数据能被传送到客户端运行的主机上并且以 CAN 数据桢的形式显示，这个主机也可以发送数据给用于转换 CAN 网络 XMOS 设备。
这段代码也可以很容易扩展成多口 CAN 转以太网桥和转换的方案，并且在 xSOFTip 库的其他系列通信协议也可以使用这个应用笔记里的原理来设计其他桥和转换。

——

需要的工具和库文件：

1. xTIMEcomposer Tools – Version 13.1.0
2. XMOS Ethernet/TCP xSOFTip component -Version 3.2.1rc1
3. XMOS CAN bus xSOFTip component - Version 2.0.0rc0

需要的软件：

这个应用笔记是基于 XMOS xCORE General Purpose(L-series) device。
这段代码被应用执行和测试在 xCORE L-series sliceKIT core board 1v2(XP-SKC-L2)上，但并不是只可以运行在这个板子上，也可以修改运行在其他的开发板上，包括：xCORE General Purpose (L-series),xCORE-USB series 或者 xCORE-Analog series 设备。

需要的知识：

1. 开发者需要熟悉 XMOS xCORE 架构，CAN 总线规格书和其他相关 CAN 规格书，XMOS 工具链和 XC 语言，附录中的列表材料并不一定针对这个应用笔记。
2. 文档中术语请参考 XMOS 术语词典注 1。
3. 使用 XMOS TCP/IP（XTCP）栈的信息请参考 Ethernet TCP/IP Component programming guide 注 2。
4. 使用 XMOS CAN 控制器的相关信息请参考 CAN Bus Controller Component documentation 注 3.

————————————————

注：

1. http://www.xmos.com/published/glossary
2. http://www.xmos.com/published/ethernet-tcpip-component-programming-guide
3. http://www.xmos.com/published/can-bus-component-%28documentation%29

1. 概述

1.1 介绍

CAN 是基于多主机协议的信息，这个信息被广播到串行总线的 CAN 网络，一个 CAN 控制器进程位来自总线和有用的应用底层 CAN 信息格式。连接一个 CAN 控制器到一个本地的网络例如以太网，提供一个有用的应用区域阵列例如：

1> CAN 总线填充；
2> 连接多重 CAN 网络；
3> 远程监测和控制。

1.2 框图

这个应用由 CAN 控制器、XTCP 栈和运行在 XMOS 设备上的执行一个 TCP 插座的服务构成。经过一个 CAN 收发器，XMOS 设备被连接在一个 CAN 总线上，结果，任意的以太网主机能连接到 XMOS 设备和收集 CAN 数据帧。

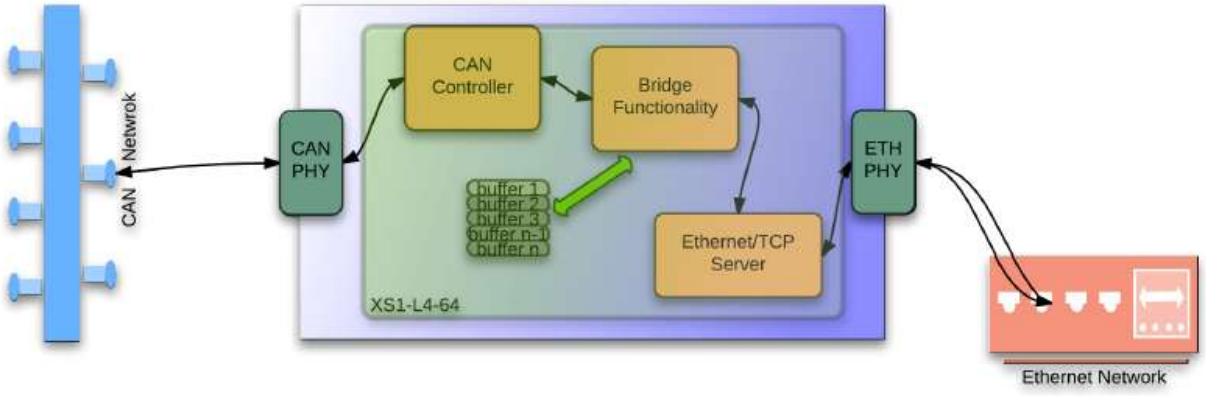

图 1：CAN 转以太网桥应用框图

2. CAN 转以太网桥应用

应用实例展示：

1> 举例一个 CAN 控制器；
2> 使用 Ethernet MII 层和 XTCP 栈作为 TCP 服务；
3> 建立一个应用任务用于桥接数据（CAN 数据帧）在一个 CAN 控制器和一个 TCP/IP 连接之间。

基于 xCORE L-series 多核控制器，执行 CAN 转换以太网桥需要 4 个任务，每个任务运行在单独的逻辑核上。

任务执行包括以下操作：

1> 一个单独的任务扮演一个 CAN 控制；
2> 两个任务执行以太网 MAC 层和 XTCP 协议层；
3> 一个任务实现应用功能：桥接 CAN 和以太网之间的数据。

这些任务通过允许在单独的逻辑核上运行并且交换数据的 xCONNECT 实现通信。

图 2 展示了 CAN 转换以太网桥的任务和通信结构的例子。

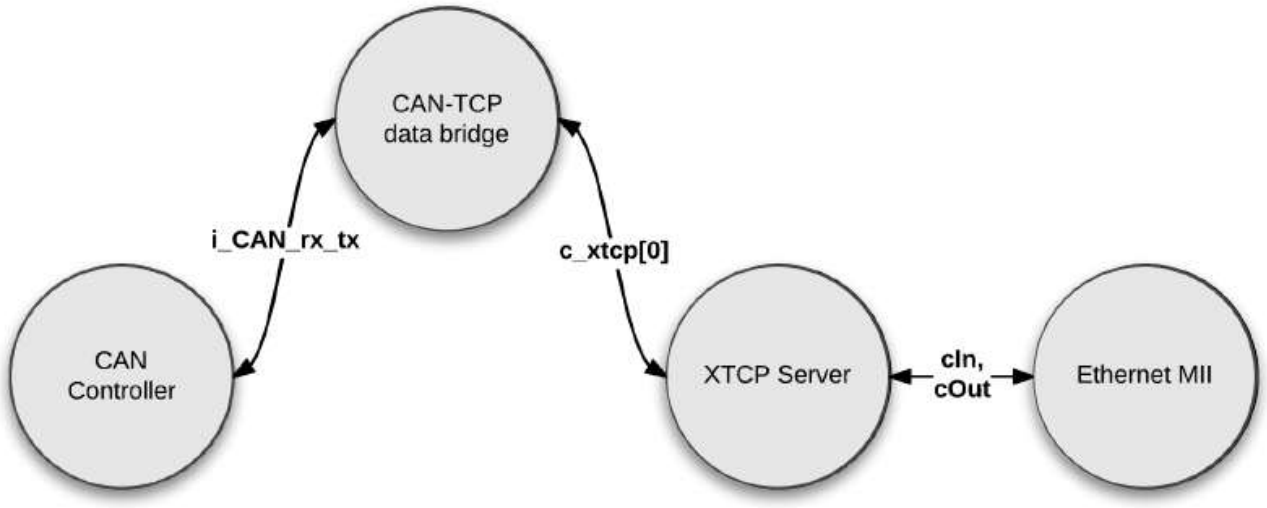

图 2：CAN 转换以太网桥的任务框图例子

2.1 应用生成文件

为了使用 XTCP 模块你需要添加模块_xtcp 和模块_can 到你的生成文件里：

```
USED_MODULES = module_xtcp module_can
```

你能使用 CAN 功能通过添加 can.h 头文件到你的源代码里

```
#include <can.h>
```

你能使用以太网和 XTCP 功能通过添加 xtcp.h 头文件到你的源代码里

```
#include <xtcp.h>
```

2.2 申明原文件和添加服务组件

Main.xc 包括 CAN 控制器和以太网/XTCP 服务脚配置，通过选择波特率、传送、接收脚、时序值，你能配置 CAN 控制器。更多的关于配置 CAN 口的信息可参考 CAN Bus API section 注 4。

```
/* CAN controller configuration */
#define BAUD_RATE_SELECTOR    2     //2 => 1000KHz, 4 => 500KHz, 8 => 250KHz, 16 => 125 Khz

can_clock_t can_clocks[NUM_CAN_IF] = {
  { BAUD_RATE_SELECTOR,
    on tile[0]:  XS1_CLKBLK_1 }
};

can_ports_t can_ports[NUM_CAN_IF] = {
// Triangle Slot
{
  on tile[0]: XS1_PORT_1I,
  on tile[0]: XS1_PORT_1L,
  8, 8, 8, 4, ACTIVE_ERROR
}
};

on tile[0]: port p_can_rs = XS1_PORT_4E;
```

注：

4：https://www.xmos.com/published/can-bus-component-%28documentation%29?secure=1

你可以配置以太网 MII 脚通过以下代码：

```
/* Ethernet configuration - Circle slot */
ethernet_xtcp_ports_t xtcp_ports = {
  on tile[1] : OTP_PORTS_INITIALIZER,
  // SMI ports
  { 0,
    on tile[1]:XS1_PORT_1H,
    on tile[1]:XS1_PORT_1G},
  // MII ports
  { on tile[1]: XS1_CLKBLK_1,
    on tile[1]: XS1_CLKBLK_2,
    on tile[1]:XS1_PORT_1J,
    on tile[1]:XS1_PORT_1P,
    on tile[1]:XS1_PORT_4E,
    on tile[1]:XS1_PORT_1K,
    on tile[1]:XS1_PORT_1I,
    on tile[1]:XS1_PORT_1L,
    on tile[1]:XS1_PORT_4F,
    on tile[1]:XS1_PORT_8B}
};
```

缺省情况下，这个应用使用 DHCP 为 XMOS 设备选择 IP 地址，如果要求一个静态的地址，你可以指定一个合法的 IP 地址。

```
#if STATIC_IP_ADDRESS
xtcp_ipconfig_t ipconfig = { { 192, 168, 2, 100 }, // ip address (eg 192,168,0,2)
  { 255, 255, 255, 0 }, // netmask (eg 255,255,255,0)
  { 192, 168, 2, 1 } // gateway (eg 192,168,0,1)
};
#else
// all 0 activates DHCP
xtcp_ipconfig_t ipconfig = { { 0,0,0,0 }, // ip address (eg 192,168,0,2)
  { 0,0,0,0}, // netmask (eg 255,255,255,0)
  { 0,0,0,0 } // gateway (eg 192,168,0,1)
};
#endif
```

基于 CAN 控制器和以太网服务的端口的定义被应用需要在 mian()中被调用。

2.3 应用 Main()功能

以下是这个应用的主函数源代码，它来自源文件 main.xc

```
int main() {
  chan c_xtcp[1];
  interface interface_can i_multi_CAN_rx_tx[NUM_CAN_IF];

  par {
      on tile[0]: {
        p_can_rs <: 0;
        par(int i=0; i<NUM_CAN_IF; i++)
          can_server(can_ports[i], can_clocks[i],i_multi_CAN_rx_tx[i]);
        }

      on tile[1]: {
        ethernet_xtcp_server(xtcp_ports, ipconfig,
              c_xtcp, 1);
       }

      on tile[1]: can_tcp_bridge(c_xtcp[0], i_multi_CAN_rx_tx);
  }

  return 0;
}
```

看到这里，更多的细节你可以看看以下内容：

1> 标准的说明描述了同时运行 3 个独立的任务；
2> 被应用的 xCONNECT 通信接口和 XTCP 被定义都在 mian 函数的最开始。
3> CAN 控制器和应用任务的通信接口的定义被安排在以上步骤之后。
4> 通过 can_server()函数调用配置和执行 CAN 控制器。
5> CAN 的时钟和接口脚配置也是通过 can_server()来实现的。
6> 通过调用 ethernet_xtcp_sever()可以配置和执行以太网和 XTCP 功能，这个函数使用两个任务，一个管理以太网 MII 层，另一个运行 xtcp 层。
7> ethernet_xtcp_sever()也需要较早的声明。
8> 通过调用和执行 can_tcp_bridge 可以实现应用功能，它主要实现在 CAN 控制器和 TCP 服务器之间的管理和桥接数据功能。

2.4 配置 CAN 控制器

CAN 控制器使用独立的收发缓冲保持 CAN 数据桢，这些被定义在 can_conf.h 里，你可以配置缓冲大小同以下代码：

```
/* Size of raw CAN message buffer */
#define CAN_FRAME_BUFFER_SIZE 15
```

2.5 配置 XTCP 服务

XTCP 栈提供设置 APIs，他是有用的对于解码收到的数据包，你可以使能 XTCP 的 APIs 功能通过设置 xtcp_client_conf.h:

```
/* Set XTCP stack to use buffered API's */
#define XTCP_BUFFERED_API 1
```

2.6 配置 CAN-TCP 桥

XTCP 服务被配置作为 TCP 接口服务，TCP 服务接口和控制 TCP 数据的缓冲长度都被定义在文件 tcp_can_protocol.h 中：

```
/* Set TCP socket server to listen to incoming connections from TCP clients */
#define TCP_CAN_PROTOCOL_PORT 15533
#define TCP_CAN_PROTOCOL_HDR_SIZE 1
/* Set the lower marker level for the transmit buffer */
#define TCP_CAN_PROTOCOL_MAX_MSG_SIZE 100
/* Set the size of TX and RX buffers */
#define TCP_CAN_PROTOCOL_RXBUF_LEN 2048
#define TCP_CAN_PROTOCOL_TXBUF_LEN 1024
```

2.7CAN-TCP 桥

设置 XTCP 栈定义是为了使用缓冲的 APIs，应用任务提供任意应用的特定收发数据 XTCP 栈缓冲，无论何时 API call 被使用，就通过 XTCP 栈更新缓冲器状态。缓冲器结构通过文件 tcp_can_protocol.h 定义：

```
/* TCP connection state */
typedef struct tcp_can_protocol_state_t
{
  int active;
  int got_header;
  int len;
  int last_used;
  int conn_id;
  xtcp_bufinfo_t bufinfo;
  char inbuf[TCP_CAN_PROTOCOL_RXBUF_LEN];
  char outbuf[TCP_CAN_PROTOCOL_TXBUF_LEN];
} tcp_can_protocol_state_t;
```

功能函数 can_tcp_bridge 是应用任务，并且它被置于 can_tcp_bridge.xc 文件中，被运行在 while(1) 循环中。其功能是等待 CAN 控制器和 XTCP 服务的数据，然后再事件循环中处理。

```
void can_tcp_bridge(chanend tcp_svr, client interface interface_can i_multi_CAN_rx_tx[NUM_CAN_IF]) {
  xtcp_connection_t conn;
  can_connection_t can_conn;
  can_frame_t can_rx_frames[NUM_CAN_IF];

  timer tmr;
  int t, timestamp;
  int conn_valid=0;
  int err;

  printf("Starting CAN to Ethernet bridge\n",NUM_CAN_IF);
  tcp_can_protocol_init(tcp_svr);
  tmr  :> t;

  // Loop forever processing CAN server and XTCP server events
  while(1) {
    select {
      case i_multi_CAN_rx_tx[int i].can_event():
        // Handle events from CAN server
```

通过 CAN 控制通知处于循环第一部分的 CAN 事件，当它从 CAN 控制接收到数据桢并且使用 xtcp_buffered_send_wrapper 函数来指示已经准备好发送数据的 XTCP 栈来发送数据。一旦这个功能触发，那么应用信息的状态被更新到缓冲并且放置一个请求信息在这个信息 TCP 包中。

```
select {
  case i_multi_CAN_rx_tx[int i].can_event():
    // Handle events from CAN server
    // Get the event generated
    can_conn = i_multi_CAN_rx_tx[i].can_get_event();
    switch(can_conn.event) {
      case E_TX_SUCCESS: {
        // Transmit of a CAN frame was successful
        break;
      }

      case E_RX_SUCCESS: {
        // A CAN frame was received from other node
        // Get the CAN frame
        err = i_multi_CAN_rx_tx[i].can_recv(can_rx_frames[i]);

        char can_to_tcp_data[NUM_CAN_BYTES]; // Host to NUM_CAN_IF Buffers
        for(int j=0; j<can_rx_frames[i].dlc; j++) {
          can_to_tcp_data[j] = can_rx_frames[i].data[j];
        }
#ifdef CAN_RX_CRC_PASSTHROUGH
        // capture CRC field
        can_to_tcp_data[8] = (char) can_rx_frames[i].crc;
        can_to_tcp_data[9] = (char) (can_rx_frames[i].crc >> 8);
#endif
        //c2t_i.can_to_tcp_message(can_to_tcp_data, NUM_CAN_BYTES);
        // Print the received frame
        //can_utils_print_frame(can_rx_frames[i], "RX: ");
        // Note: CAN data will be dropped here until connection is valid
        if(conn_valid) {
#if CAN_RX_CRC_PASSTHROUGH > 0
          int success = xtcp_buffered_send_wrapper(tcp_svr, conn, can_to_tcp_data, NUM_CAN_BYTES);
```

一旦 XTCP 识别了这个请求，应用任务就能使用放在文件 tcp_can_protocol_tranport.c 中的函数 tcp_can_protocol_send 来发送数据，这个函数发送连接应用数据的缓冲。

```
void tcp_can_protocol_send(chanend tcp_svr, xtcp_connection_t *conn) {
  struct tcp_can_protocol_state_t *st =
    (struct tcp_can_protocol_state_t *) conn->appstate;
  xtcp_buffered_send_handler(tcp_svr, conn, &st->bufinfo);
  return;
```

第二部分为主控 XTCP 服务事件的函数 can_tcp_bridge，一旦 XTCP 服务有任何动作，这个函数就调用 tcp_can_protocol_handle_event 功能函数。

```
case xtcp_event(tcp_svr, conn): {
  // Send the event to the protocol handler for processing
  tmr :> timestamp;
  switch (conn.event) {
    case XTCP_NEW_CONNECTION:
      conn_valid = 1;
      break;
    case XTCP_IFDOWN:
    case XTCP_TIMED_OUT:
    case XTCP_ABORTED:
    case XTCP_CLOSED:
      conn_valid = 0;
      break;
    default:
      break;
  }
  tcp_can_protocol_handle_event(tcp_svr, conn, timestamp, i_multi_CAN_rx_tx);
```

2.8 TCP 事件管理器

在文件 tcp_can_protocol_transport.c 中定义了 tcp_can_protocol_handle_event 功能函数。

```
void tcp_can_protocol_handle_event(chanend tcp_svr, xtcp_connection_t *conn,
        int timestamp, unsigned i_multi_CAN_rx_tx[]) {
  // We have received an event from the TCP stack, so respond
  // appropriately

  // Ignore events that are not directly relevant to us
  switch (conn->event) {
```

一旦从 XTCP 服务有新的客户端连接，通过被初始化的上面段落描述的缓冲就被定义。通过 XTCP 被定义的缓冲被 XTCP 缓冲 API 功能调用。

```
 * This associates some buffer state with a conenction.  It uses three
 * API calls.
 *
 * xtcp_set_connection_appstate - for attaching a piece of user state
 * to a xtcp connection.
 *
 * xtcp_buffered_set_rx_buffer - to associate the rx memory buffer with a
 * buffered connection.
 *
 * xtcp_buffered_set_tx_buffer - to associate the tx memory buffer with a
 * buffered connection.
 */
static void tcp_can_protocol_init_state(chanend tcp_svr, xtcp_connection_t *conn,
  int timestamp) {
  int i;

  for (i = 0; i < NUM_TCP_CAN_PROTOCOL_CONNECTIONS; i++) {
    if (!connection_states[i].active)
        break;
  }

  if (i == NUM_TCP_CAN_PROTOCOL_CONNECTIONS)
    xtcp_abort(tcp_svr, conn);
  else {
    connection_states[i].active = 1;
    connection_states[i].got_header = 0;
    connection_states[i].last_used = timestamp;
    connection_states[i].conn_id = conn->id;
    xtcp_set_connection_appstate(tcp_svr, conn,
            (xtcp_appstate_t) &connection_states[i]);
    xtcp_buffered_set_rx_buffer(tcp_svr, conn,
            &connection_states[i].bufinfo, connection_states[i].inbuf,
            TCP_CAN_PROTOCOL_RXBUF_LEN);
    xtcp_buffered_set_tx_buffer(tcp_svr, conn,
            &connection_states[i].bufinfo, connection_states[i].outbuf,
            TCP_CAN_PROTOCOL_TXBUF_LEN, TCP_CAN_PROTOCOL_MAX_MSG_SIZE);
  }
  return;
```

当服务 XTCP 收到 TCP 数据的时候，事件 XTCP_RECV_DATA 就被通知给应用任务。应用任务就确认数据并且发送这个数据给 CAN 控制器，被调用功能如下：

```
case XTCP_RECV_DATA:
  if (st) {
    st->last_used = timestamp;
    tcp_can_protocol_recv(tcp_svr, conn, i_multi_CAN_rx_tx);
  } else
```

功能函数 tcp_can_protocol_recv 发送整个 tcp 包。

```
void tcp_can_protocol_recv(chanend tcp_svr, xtcp_connection_t *conn, unsigned i_multi_CAN_rx_tx[]) {
  struct tcp_can_protocol_state_t *st =
      (struct tcp_can_protocol_state_t *) conn->appstate;
  int len = 0;
  // keep looping until we have received all of the data
  do {
    // if we have not already seen and processed the header then receive a header's worth
    // of data
    if (!st->got_header) {
      char *hdr;
      int overflow = 0;

      // read the header
      len = xtcp_buffered_recv(tcp_svr, conn, &st->bufinfo, &hdr,
              TCP_CAN_PROTOCOL_HDR_SIZE, &overflow);
      if (overflow) {
        xtcp_abort(tcp_svr, conn);
```

最终，存放在 tcp_can_protocol.xc 文件中功能函数 tcp_can_protocol_process_message 加工处理这个信息。函数功能收到上面的 TCP 数据以及 CAN 数据桢，通过 can_send 功能发送这个信息给 CAN 控制器。

```
void tcp_can_protocol_process_message(client interface interface_can i_multi_CAN_rx_tx[NUM_CAN_IF], char msg
  ↪ [], int len) {
  int err;

  can_frame_t f;
  f.dlc = 8;
  f.remote = 0;
  f.extended  = 1;

#if CHECK_TCP_CAN_PROTOCOL
  if((len % (NUM_CAN_BYTES*NUM_CAN_IF)) != 0) {
    printf("Invalid data length in packet: %d. len must be a multiple of %d\n",len, NUM_CAN_BYTES*NUM_CAN_IF);
    assert(0);
  }
#endif

  // For each CAN Interface:
  // Assign received TCP data to a CAN frame and send it to the respective can_server via the can_interface
    ↪ array
  for(int base=0; base<len; base+=NUM_CAN_BYTES*NUM_CAN_IF) {
    for(int i=0; i < NUM_CAN_IF; i++) {
      for(int j=0; j<8; j++) {
        f.data[j] = msg[base+i*NUM_CAN_BYTES+j];
      }

#ifdef CAN_TX_CRC_PASSTHROUGH
      // assitn
      f[i].crc = msg[i*NUM_CAN_BYTES+8];
      f[i].crc |= (msg[i*NUM_CAN_BYTES+9] << 8);
#endif

      err = i_multi_CAN_rx_tx[i].can_send(f);
#ifdef CAN_DEBUG
      can_utils_print_frame(f, "TX: ");
#endif
    }
  }
```

小结一下，你可以看到：

1> 在应用中,CAN 控制器和 XTCP 服务模块是如何被实例化。

2> 应用中的任务缓冲是如何使用 XTCP 加工发送和接收 TCP 数据。

3> 在 CAN 控制器和 XTCP 服务中，在应用中通过不同的功能处理各种各样的数据事件，以及如何在它们之间桥接数据。

附录 A---Demo 硬件安装

为了在你的 Demo 上安装一个 CAN 网络，你需要一个 USB 转 CAN 的电子狗，连接到你的 PC 平台开发环境，Demo 的一部分你可以从以下网站获得 http://www.cananalyser.co.uk/ ，XMOS 设备被连接 CAN 节点在 PC 开发环境中。

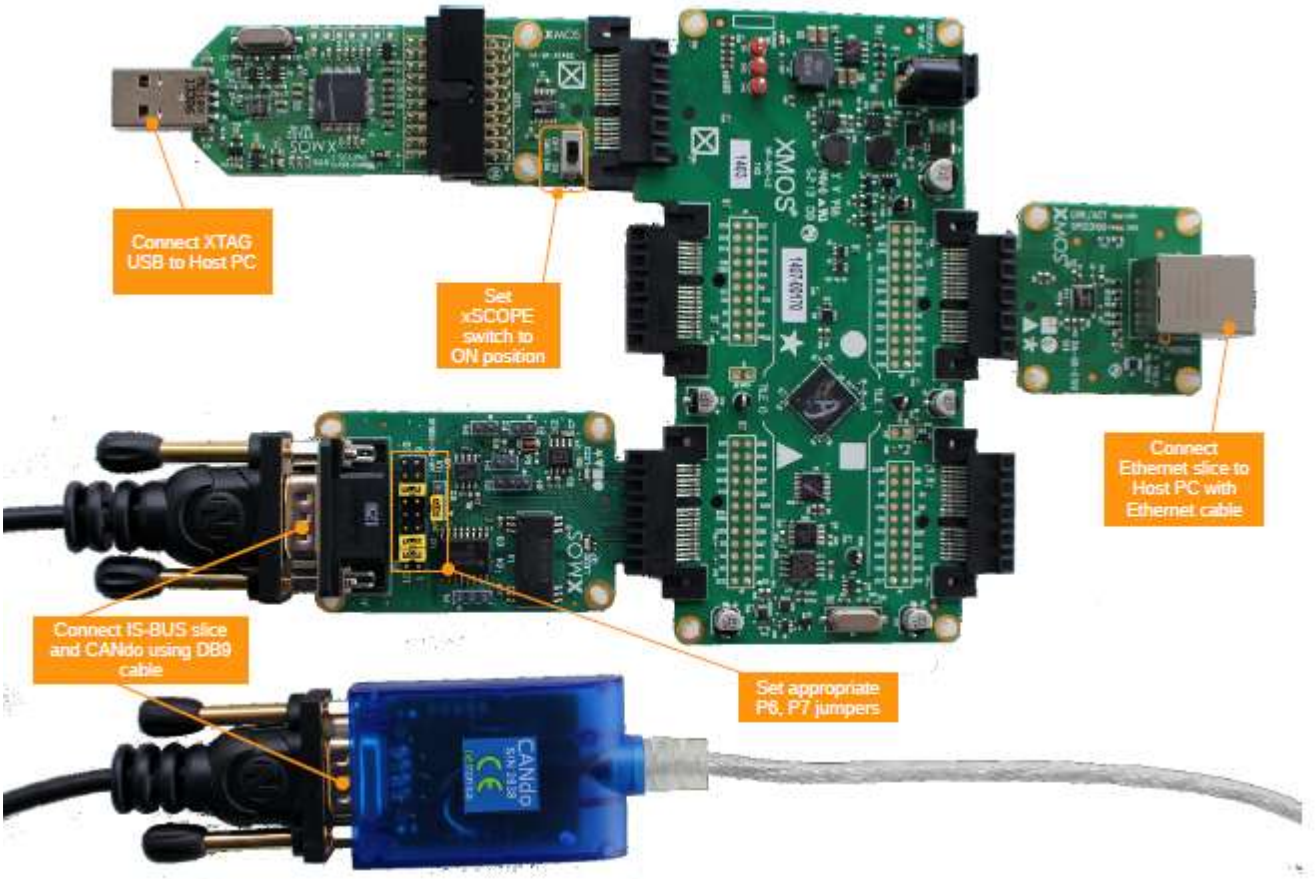

图 3 XMOS xCORE-L16 sliceKIT 安装 CAN 转以太网桥

为了运行 Demo，请完成下列操作：

1. 使用连接器连接 IS-BUS slice（XA-SK-ISBUS）和 sliceKIT core 板；
2. 设置 IS-BUS slice 上的跳线到 CAN 模式：短接冒 P7 连接 pin 1 和 2（pin 3 不连接），短接冒 P6 连接 2 和 11，3 和 12，7 和 16；
3. 连接 IS-BUS slice 到 USB 转 CAN 接口，经过 DB9 线缆；
4. 使用连接器连接以太网 slice（XA-SK-E100）和 sliceKIT core 板。
5. 连接以太网线缆，以端在以太网 slice ，另一端连接到电脑的 RJ45 接口。
6. 连接小 xTAG-2 调试器。
7. 连接调试器到你的 PC 上。
8. 设置 XMOS 环境连接到调试器。
9. 打开连接到 sliceKIT core 板的电源。

注：这个例子在这里是使用 windows PC 来安装的，但是它也可以在 limux 系统上运行。在你尝试连接 Demo 之前，你必须安装驱动和 USB 转 CAN 节点 UI 到你的 linux 上。

A.1 Windows Host

1. 安装 CANdo 到你的 window 电脑，连接 CANdo 到你的 PC 环境。
2. CANdo 应用设置

1> 点击安装 CANdo 浮标，设置波特率为 1M。
2> 点击 view→option then ensure the Display On CAN View Page option is checked.然后点击 OK。
3> 切换到 CAN view 标签。
4> 点击绿色（开始 CANdo）按钮。

3. 安装 python（版本 2.6 及以上）在你的 windows 电脑上。

附录 B 运行 Demo 设备

只要你建立了 Demo 例程，不管是使用命令行 xmake 还是 xTIMEcomposer studio 软件，你就能在 SliceKIT core 板上运行应用。

只要工程被建立，包括库函数，xCORE 内核等资源的引用目录就被确定了。

B.1 运行命令行

根据命令行，使用 xrool 工具下载代码到 xmos 设备中。指定工程的目录和在微处理器中运行代码如下：

```
> xrun --xscope can_ethernet_bridge_example.xe          <-- Download and execute the xCORE code
```

只要命令运行了，你就可以看到在 windows 显示以下文本：

```
Starting CAN to Ethernet bridge
Address: 0.0.0.0
Gateway: 0.0.0.0
Netmask: 0.0.0.0
ipv4ll: 169.254.190.19
```

B.2 从 xTIMEmposer studio 运行例子

通过 xTIMEmposer studio 进程，下载代码到 XMOS 设备中，在 bin 目录下选择 xCORE 二进制文件，右键点击如下选择：

1. 选择运行配置。
2. 使能板子 I/O 选项的 xSCOPE。

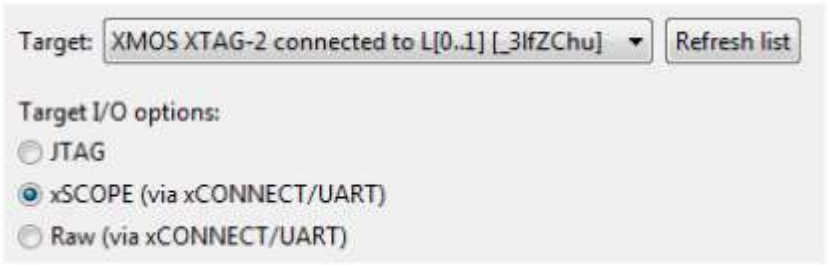

图 4 xTIMEmposer studio 配置

3. 点击应用和运行。

当进程完成引导，你会在 xTIMEmposer console 看到以下文本：

```
Starting CAN to Ethernet bridge
Address: 0.0.0.0
Gateway: 0.0.0.0
Netmask: 0.0.0.0
ipv4ll: 169.254.190.19
```

附录 C 运行 Demo

C.1 Windows 主机

1. 选择 CANdo 应用，确保 CANdo 接口成功的接入系统。
2. 打开一个控制台，执行一个主机测试（一个 TCP 包客户端）：

```
python C:\can_ethernet_bridge_example\test\test_can_ethernet_bridge.py 169.254.190.19
```

注意：确保你提供了正确的源代码脚本目录和以及 XMOS 设备的 IP 地址显示在 xTIMEcomposer 控制台界面。

3. 只要你的主机测试脚本连接上 XMOS 设备，就会有如下显示：

```
Connecting..
Connected socket: IP 169.254.190.19, Port 15533
```

4. 主机测试脚本会发送数据包（5X8 的 CAN 数据包作为测试数据）到 XMOS 设备如下：

```
Will now send 1 Packets with 5 CAN data blocks (40 bytes) per packet...
Msg to send: (XMOS0000XMOS1111XMOS2222XMOS3333XMOS4444
```

5. 在 CANdo 应用上选择 CAN view tab，检查显示的 RX 帧时候含有主机测试脚本发送的数据，如果没有显示，请检查连接和安装时候完整，并且重新连接 Demo。
6. 发送相同数据从 CANdo，XMOS 设备接收这个 CAN 帧并且这个帧发送到主机上。
7. 主机脚本接收到 CAN 帧并且显示在控制台上：

```
Receiving message from CAN over TCP: ['X', 'M', 'O', 'S', '4', '4', '4', '4']
```

附录 D 参考文献

XMOS Tools User Guide
http://www.xmos.com/published/xtimecomposer-user-guide
XMOS xCORE Programming Guide
http://www.xmos.com/published/xmos-programming-guide
XMOS Layer 2 Ethernet MAC Component
https://www.xmos.com/published/xmos-layer-2-ethernet-mac-component
CAN Specification Version 2.0
http://www.bosch-semiconductors.de/media/pdf_1/canliteratur/can2spec.pdf

附录 E 全部的源代码列表

E.1 Soure code for main.xc

```
#include <platform.h>
#include <xs1.h>
#include "xtcp.h"
#include "otp_board_info.h"
#include "can.h"
#include "can_conf.h"
#include "can_util.h"
#include "xtcp.h"
#include "xscope.h"
#include "stdio.h"
#include "tcp_server.h"
#include <stdio.h>
#include <print.h>

/*---------------------------------------------------------------------------
Macros
---------------------------------------------------------------------------*/
/* CAN controller configuration */
#define BAUD_RATE_SELECTOR    2     //2 => 1000KHz, 4 => 500KHz, 8 => 250KHz, 16 => 125 Khz

can_clock_t can_clocks[NUM_CAN_IF] = {
  { BAUD_RATE_SELECTOR,
    on tile[0]:  XS1_CLKBLK_1 }
};

can_ports_t can_ports[NUM_CAN_IF] = {
// Triangle Slot
{
  on tile[0]: XS1_PORT_1I,
  on tile[0]: XS1_PORT_1L,
  8, 8, 8, 4, ACTIVE_ERROR
}
};
```

```
on tile[0]: port p_can_rs = XS1_PORT_4E;

/* Ethernet configuration - Circle slot */
ethernet_xtcp_ports_t xtcp_ports = {
  on tile[1] : OTP_PORTS_INITIALIZER,
  // SMI ports
  { 0,
    on tile[1]:XS1_PORT_1H,
    on tile[1]:XS1_PORT_1G},
  // MII ports
  { on tile[1]: XS1_CLKBLK_1,
    on tile[1]: XS1_CLKBLK_2,
    on tile[1]:XS1_PORT_1J,
    on tile[1]:XS1_PORT_1P,
    on tile[1]:XS1_PORT_4E,
    on tile[1]:XS1_PORT_1K,
    on tile[1]:XS1_PORT_1I,
    on tile[1]:XS1_PORT_1L,
    on tile[1]:XS1_PORT_4F,
    on tile[1]:XS1_PORT_8B}
};
```

```
#if STATIC_IP_ADDRESS
xtcp_ipconfig_t ipconfig = { { 192, 168, 2, 100 }, // ip address (eg 192,168,0,2)
  { 255, 255, 255, 0 }, // netmask (eg 255,255,255,0)
  { 192, 168, 2, 1 } // gateway (eg 192,168,0,1)
};
#else
// all 0 activates DHCP
xtcp_ipconfig_t ipconfig = { { 0,0,0,0 }, // ip address (eg 192,168,0,2)
  { 0,0,0,0}, // netmask (eg 255,255,255,0)
  { 0,0,0,0 } // gateway (eg 192,168,0,1)
};
#endif
```

```
/* xSCOPE Setup Function */
void xscope_user_init(void) {
  xscope_register(0, 0, "", 0, "");
  xscope_config_io(XSCOPE_IO_BASIC);
}

int main() {
  chan c_xtcp[1];
  interface interface_can i_multi_CAN_rx_tx[NUM_CAN_IF];

  par {
      on tile[0]: {
        p_can_rs <: 0;
        par(int i=0; i<NUM_CAN_IF; i++)
          can_server(can_ports[i], can_clocks[i],i_multi_CAN_rx_tx[i]);
        }

      on tile[1]: {
        ethernet_xtcp_server(xtcp_ports, ipconfig,
              c_xtcp, 1);
       }

      on tile[1]: can_tcp_bridge(c_xtcp[0], i_multi_CAN_rx_tx);
  }

  return 0;
}
```

E.2 Source code for can_tcp_bridge.xc

```
// Copyright (c) 2014, XMOS Ltd, All rights reserved
// This software is freely distributable under a derivative of the
// University of Illinois/NCSA Open Source License posted in
// LICENSE.txt and at <http://github.xcore.com/>

#include <xs1.h>
#include <print.h>
#include "xtcp_client.h"
#include "xtcp_buffered_client.h"
#include "tcp_server.h"
#include <string.h>
#include <assert.h>
#include <can.h>
#include <can_conf.h>
#include <can_util.h>
#include <stdio.h>

#define TCP_CAN_PROTOCOL_PERIOD_MS (200) // every 200 ms
#define TCP_CAN_PROTOCOL_PERIOD_TIMER_TICKS  (TCP_CAN_PROTOCOL_PERIOD_MS * XS1_TIMER_KHZ)

// just for debug
char local_msg[NUM_CAN_BYTES];

// The main service thread
void can_tcp_bridge(chanend tcp_svr, client interface interface_can i_multi_CAN_rx_tx[NUM_CAN_IF]) {
  xtcp_connection_t conn;
  can_connection_t can_conn;
  can_frame_t can_rx_frames[NUM_CAN_IF];
```

```
  timer tmr;
  int t, timestamp;
  int conn_valid=0;
  int err;

  printf("Starting CAN to Ethernet bridge\n",NUM_CAN_IF);
  tcp_can_protocol_init(tcp_svr);
  tmr   :> t;

  // Loop forever processing CAN server and XTCP server events
  while(1) {
    select {
      case i_multi_CAN_rx_tx[int i].can_event():
```

```
        // Handle events from CAN server
        // Get the event generated
        can_conn = i_multi_CAN_rx_tx[i].can_get_event();
        switch(can_conn.event) {
          case E_TX_SUCCESS: {
            // Transmit of a CAN frame was successful
            break;
          }

          case E_RX_SUCCESS: {
            // A CAN frame was received from other node
            // Get the CAN frame
            err = i_multi_CAN_rx_tx[i].can_recv(can_rx_frames[i]);

            char can_to_tcp_data[NUM_CAN_BYTES]; // Host to NUM_CAN_IF Buffers
            for(int j=0; j<can_rx_frames[i].dlc; j++) {
              can_to_tcp_data[j] = can_rx_frames[i].data[j];
            }
#ifdef CAN_RX_CRC_PASSTHROUGH
            // capture CRC field
            can_to_tcp_data[8] = (char) can_rx_frames[i].crc;
            can_to_tcp_data[9] = (char) (can_rx_frames[i].crc >> 8);
#endif
            //c2t_i.can_to_tcp_message(can_to_tcp_data, NUM_CAN_BYTES);
            // Print the received frame
            //can_utils_print_frame(can_rx_frames[i], "RX: ");
            // Note: CAN data will be dropped here until connection is valid
            if(conn_valid) {
#if CAN_RX_CRC_PASSTHROUGH > 0
              int success = xtcp_buffered_send_wrapper(tcp_svr, conn, can_to_tcp_data, NUM_CAN_BYTES);
```

```
#else
#if BRIDGE_FULL_CAN_FRAME > 0
              int success = xtcp_buffered_send_wrapper(tcp_svr, conn, (char *) &can_rx_frames[i], sizeof(
                ↪ can_frame_t));
#else
              int success = xtcp_buffered_send_wrapper(tcp_svr, conn, can_to_tcp_data, can_rx_frames[i].dlc);
#endif //BRIDGE_FULL_CAN_FRAME
#endif //CAN_RX_CRC_PASSTHROUGH
              if (!success)
                  printstr("send buffer overflow\n");
            }
            break;
          }

          case E_BIT_ERROR:
          case E_STUFF_ERROR:
          case E_ACK_ERROR:
          case E_FORM_ERROR:
          case E_CRC_ERROR: {
            // There was an error while RX or TX frame
              printstr("state = "); printintln(can_conn.state);
              printstr("event = "); printintln(can_conn.event);
              printstr("TEC = "); printintln(can_conn.tec);
              printstr("REC = "); printintln(can_conn.rec);
              printstrln("----------------------------");
              break;
          }

          default: break;
        } // switch(conn.event)
        break;
```

```
          }

          default: break;
        } // switch(conn.event)
        break;

        case xtcp_event(tcp_svr, conn): {
          // Send the event to the protocol handler for processing
          tmr :> timestamp;
          switch (conn.event) {
            case XTCP_NEW_CONNECTION:
              conn_valid = 1;
              break;
```

```
            case XTCP_IFDOWN:
            case XTCP_TIMED_OUT:
            case XTCP_ABORTED:
            case XTCP_CLOSED:
              conn_valid = 0;
              break;
```

```
            default:
              break;
          }
          tcp_can_protocol_handle_event(tcp_svr, conn, timestamp, i_multi_CAN_rx_tx);
        }
        break;

        case tmr when timerafter(t) :> void: {
          // Send a periodic event to the protocol
          tcp_can_protocol_periodic(tcp_svr, t);
          t += TCP_CAN_PROTOCOL_PERIOD_TIMER_TICKS;
         }
        break;
    } //end of select
  }
}
```

E.3 源代码文件：tcp_can_protocol.xc

```
// Copyright (c) 2014, XMOS Ltd, All rights reserved
// This software is freely distributable under a derivative of the
// University of Illinois/NCSA Open Source License posted in
// LICENSE.txt and at <http://github.xcore.com/>

#include "xtcp_client.h"
#include "xtcp_buffered_client.h"
#include "tcp_can_protocol.h"
#include "stdio.h"
#include "assert.h"
#include "can.h"
#include "can_conf.h"
#include "can_util.h"

#define CHECK_TCP_CAN_PROTOCOL 1

// This method is called by the protocol RX function when a full message is received
void tcp_can_protocol_process_message(client interface interface_can i_multi_CAN_rx_tx[NUM_CAN_IF], char msg
  ↪ [], int len) {
  int err;

  can_frame_t f;
  f.dlc = 8;
  f.remote = 0;
  f.extended  = 1;

#if CHECK_TCP_CAN_PROTOCOL
  if((len % (NUM_CAN_BYTES*NUM_CAN_IF)) != 0) {
    printf("Invalid data length in packet: %d. len must be a multiple of %d\n",len, NUM_CAN_BYTES*NUM_CAN_IF);
    assert(0);
  }
#endif
```

```
  // For each CAN Interface:
  // Assign received TCP data to a CAN frame and send it to the respective can_server via the can_interface
    ↪ array
  for(int base=0; base<len; base+=NUM_CAN_BYTES*NUM_CAN_IF) {
    for(int i=0; i < NUM_CAN_IF; i++) {
      for(int j=0; j<8; j++) {
        f.data[j] = msg[base+i*NUM_CAN_BYTES+j];
      }

#ifdef CAN_TX_CRC_PASSTHROUGH
      // assitn
      f[i].crc = msg[i*NUM_CAN_BYTES+8];
      f[i].crc |= (msg[i*NUM_CAN_BYTES+9] << 8);
#endif
```

```
      err = i_multi_CAN_rx_tx[i].can_send(f);
#ifdef CAN_DEBUG
      can_utils_print_frame(f, "TX: ");
#endif
    }
  }
```

```
  return;
}
```

茂晶有限公司 GFE International (Hong Kong) Limited A MACNICA Company

XMOS 特許一級代理商

茂晶有限公司
國內服務據點 ：
深圳：Tel: 86-755-8828 5788-1207 Email: Shenzhen@gfei.com.hk
北京：Tel: 86-10-5126 6624 Email: Beijing@gfei.com.hk
上海：Tel: 86-21-54453155 Email: Shanghai@gfei.com.hk
武漢：Tel: 86-27-8730 6822, 8784 0783 Email: Wuhan@gfei.com.hk
青島：Tel: 86-532-8573 1420 Email: Qingdao@gfei.com.hk
成都：Tel: 86-028-85548390 Email: Chengdu@gfei.com.hk
廈門：Tel: 86-0592-5302668 Email: Xiamen@gfei.com.hk
海外服務據點 ：
香港：Tel: 852-3741 0662-2293 Email: Hongkong@gfei.com.hk
台灣：Tel: 886-2-89132200 Email: Service@gfei.com.hk
技術支援 ：
深圳：Tel: 86-755-8828 5788-1207 Email: royl@gfei.com.hk
上海：Tel: 86-21-54453155
北京：Tel: 86-10-8429 8668 Email: jackl@gfei.com.hk