Application Note: AN00114

CAN to Ethernet bridge

Since XMOS multicore microcontrollers implement communications protocols purely in software, very flexible bridging systems can be built that meet your exact requirements.

This application note shows how to create a single port CAN protocol to an Ethernet bridge. An application bridge function is created to demonstrate how CAN frames from the CAN network are converted into Ethernet (TCP packets) and vice versa. The Ethernet data from the XMOS device can be sent to a TCP client application running on the host which displays them as CAN frames on the console. This host application can also be used to send data to the XMOS device which in turn is sent to the CAN network.

This code could easily be extended to build multi-port CAN to Ethernet bridge and switch solutions. Furthermore, the principles demonstrated in this application note can be used to build bridges and switches for any of the array of communications protocols supported in xSOFTip libraries.

Required tools and libraries

- xTIMEcomposer Tools - Version 13.1.0
- XMOS Ethernet/TCP xSOFTip component - Version 3.2.1rc1
- XMOS CAN bus xSOFTip component - Version 2.0.0rc0

Required hardware

This application note is designed to run on an XMOS xCORE General Purpose (L-series) device.

The example code provided with the application has been implemented and tested on the xCORE L-series sliceKIT core board 1V2 (XP-SKC-L2) but there is no dependency on this board and it can be modified to run on any development board which uses an xCORE General Purpose (L-series), xCORE-USB series or xCORE-Analog series device.

Prerequisites

- This document assumes familiarity with the XMOS xCORE architecture, the Controller Area Network (CAN) bus specification (and related specifications), the XMOS tool chain and the xC language. Documentation that is not specific to this application note is listed in the references appendix.
- For descriptions of XMOS related terms found in this document please see the XMOS Glossary[1].
- For information on using XMOS TCP/IP (XTCP) stack please see the *Ethernet TCP/IP component programming guide*[2].
- For information on using XMOS CAN controller please see the *CAN Bus Controller Component* documentation[3].

[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 Overview

1.1 Introduction

CAN is a message based multi-master protocol. The messages are broadcast in the serial bus CAN network. A CAN controller processes bits from the bus and forms the CAN messages useful for the application layers. Connecting a CAN controller to a local network such as Ethernet provides an array of useful application areas such as:

- CAN bus extenders
- Linking multiple CAN networks
- Remote monitoring and control

1.2 Block diagram

This application uses the CAN controller and the XTCP stack and implements a TCP socket server running on the XMOS device. The XMOS device is connected to a CAN bus to participate in the CAN network via a CAN transceiver. As a result, any Ethernet host (acting as a TCP client) can connect to this XMOS device and collect the CAN frames.

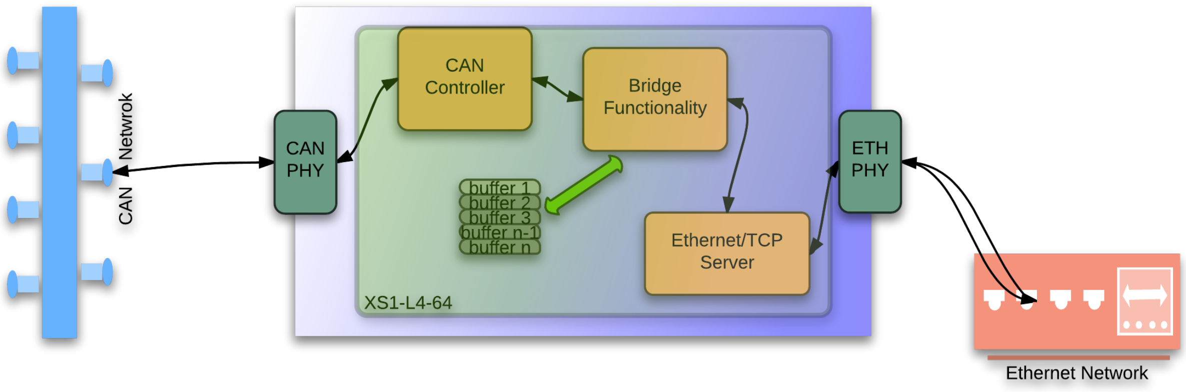


Figure 1: Block diagram of CAN to Ethernet bridge application example

2 CAN to Ethernet bridge application

This application demonstrates how to:

- instantiate a CAN controller
- use Ethernet MII layer and XTCP stack as a TCP server
- build an application task to bridge the data (CAN frames) between a CAN controller and a TCP/IP connection

To implement a CAN to Ethernet bridge requires four tasks running on separate logical cores of an xCORE L-series multicore microcontroller.

The tasks perform the following operations:

- A single task acting as a CAN controller
- Two tasks implementing Ethernet MAC layer and XTCP protocol stack
- A task implementing the application functionality to bridge the data between CAN and Ethernet networks

These tasks communicate via the use of xCONNECT channels which allow the data to be passed between the code running on separate logical cores.

Figure 2 shows the task and communication structure for this CAN to Ethernet bridge example.

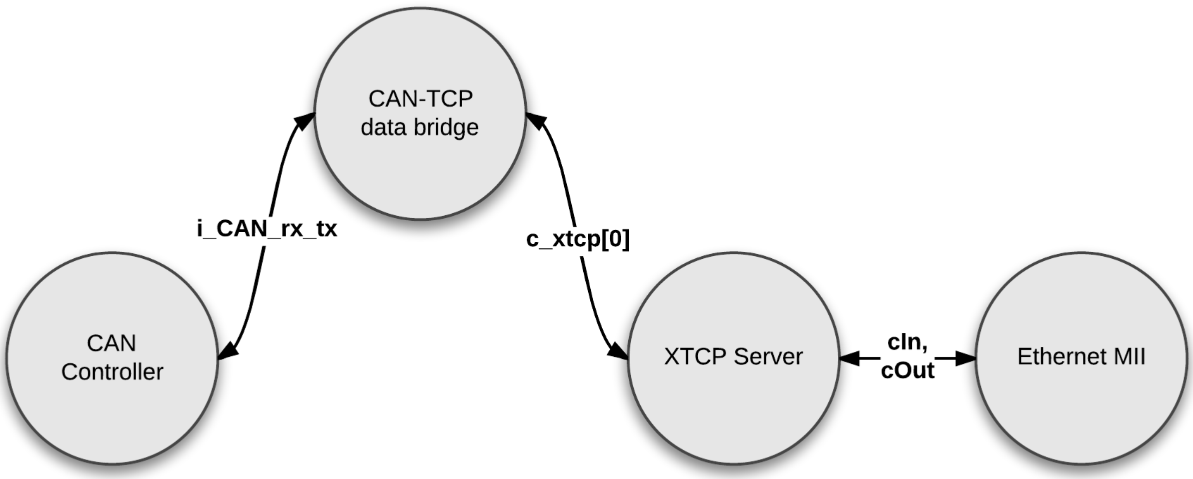


Figure 2: Task diagram of CAN to Ethernet bridge example

2.1 Makefile additions for this application

To start using the XTCP module you need to add module_xtcp and module_can to your makefile:

```
USED_MODULES = module_xtcp module_can
```

You can access the CAN functions in your source code via the can.h header file:

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

You can access the Ethernet and XTCP functions in your source code via the xtcp.h header file:

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

2.2 Declaring resources and setting up the server components

main.xc contains the pin configurations for the CAN controller and Ethernet/XTCP server. You can configure the CAN controller by selecting the baud rate, transmit, receive pins and the bit timing values. More information on the can_ports[] configuration settings is available in CAN Bus API section[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;
```

[4] http://www.xmos.com/published/can-bus-component-%28documentation%29

You can configure the Ethernet MII pins as follows:

```
/* 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}
};
```

By default, the application uses DHCP to select an IP address for the XMOS device. If a static address is required then you need to specify a valid IP address.

```
#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
```

The above port definitions are passed to the functions of the CAN controller and Ethernet/XTCP server which are called from `main()`.

2.3 The application main() function

Below is the source code for the main function of this application, which is taken from the source file `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;
}
```

Looking at this in more detail you can see the following:

- The par statement describes running three separate tasks in parallel
- The xCONNECT communication channel used by the application and XTCP server is set up at the beginning of main()
- The interface declaration to communicate between the CAN controller and application task is set up next in main()
- There is a can_server() function call to configure and execute the CAN controller
- The CAN clock and port (pin) configuration discussed earlier is passed to the function can_server()
- There is a function call to configure and execute the Ethernet and XTCP server: ethernet_xtcp_server(). This function spawns two tasks; one for managing the Ethernet MII layer and another that runs the XTCP stack
- The Ethernet port configuration discussed earlier is passed to the function ethernet_xtcp_server()
- There is a can_tcp_bridge() function call to implement the application functionality. This function primarily manages and bridges the data between the CAN controller and TCP server

2.4 Configuring the CAN controller

The CAN controller uses separate transmit and receive buffers to hold the CAN frames. These are defined in the file can_conf.h. You can configure the buffer size as follows:

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

2.5 Configuring the XTCP server

XTCP stack provides a set of APIs (called buffered mode APIs) which can be useful if the application protocol is known and needs to be operated to decode the received packets as and when they are received. You can enable the XTCP stack to use the buffered mode APIs by setting the following from the file xtcp_client_conf.h:

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

2.6 Configuring the CAN-TCP bridge

XTCP server is configured to act as a TCP socket server. The TCP server port and the length of buffers to handle the TCP data are defined in the file `tcp_can_protocol.h`.

```
/* 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
```

2.7 CAN-TCP bridge

As the XTCP stack is configured to use buffered APIs, the application task provides the buffers for the XTCP stack for any application specific incoming or outgoing data. The buffer state is updated by the XTCP stack whenever corresponding API call is made. The buffer structure is defined in the file `tcp_can_protocol.h`.

```
/* 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;
```

The function `can_tcp_bridge` is the application task. It is implemented in the file `can_tcp_bridge.xc` and runs a `while (1)` loop. This function waits for data from CAN controller and XTCP server and process it using an event loop.

```
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
```

The first part of the event loop handles CAN events as notified by the CAN controller. It receives frames from the CAN controller and uses a function `xtcp_buffered_send_wrapper` to indicate to the XTCP stack that the application task is ready to send some data. Whenever this function is invoked, the XTCP server copies the application message in to the corresponding application state buffer and places a request to transmit this message as a TCP packet.

```
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);
```

Once the XTCP server acknowledges this request, the application task can send the data using the function `tcp_can_protocol_send` as defined in the file `tcp_can_protocol_transport.c`. This function passes the buffer details of the connection to send the application data.

```
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;
```

The second part of the function `can_tcp_bridge` handles the events from the XTCP server. It calls the function *tcp_can_protocol_handle_event* whenever there is any activity related to the XTCP server.

```
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 event handler

The function `tcp_can_protocol_handle_event` is defined in the file `tcp_can_protocol_transport.c`.

```
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) {
```

The application defined buffers that are described in the above sections are initialized whenever there is any new client connection event from the XTCP server. The XTCP buffered API functions are called to use the application defined buffers by the XTCP server.

```
 * 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;
```

When TCP data is received by XTCP server, an XTCP_RECV_DATA event is notified to the application task. The application task validates the data and sends this data to the CAN controller. The function call sequence is displayed below:

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

The function `tcp_can_protocol_recv` receives the entire TCP packet.

```
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);
```

Finally, the message is processed by the function `tcp_can_protocol_process_message` as defined in the file `tcp_can_protocol.xc`. This function receives the above TCP data and forms a CAN frame and sends it to the CAN controller using the can_send function.

```
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
    }
  }
```

In summary, you have seen:

- how the CAN controller and XTCP server modules are instantiated in the application
- how the application task buffers are used by XTCP server to process the incoming and outgoing TCP data
- the different functions that are used in the application task to handle various data events from the CAN controller and the XTCP server, and how data is bridged between them

APPENDIX A - Demo hardware setup

In order to set up a CAN network for this demo, you need a CAN to USB dongle connected to your development PC (Windows platform). The one used for this demo is available from http://www.cananalyser.co.uk. The XMOS device (acting as a CAN node) is connected to the CAN node on the development PC.

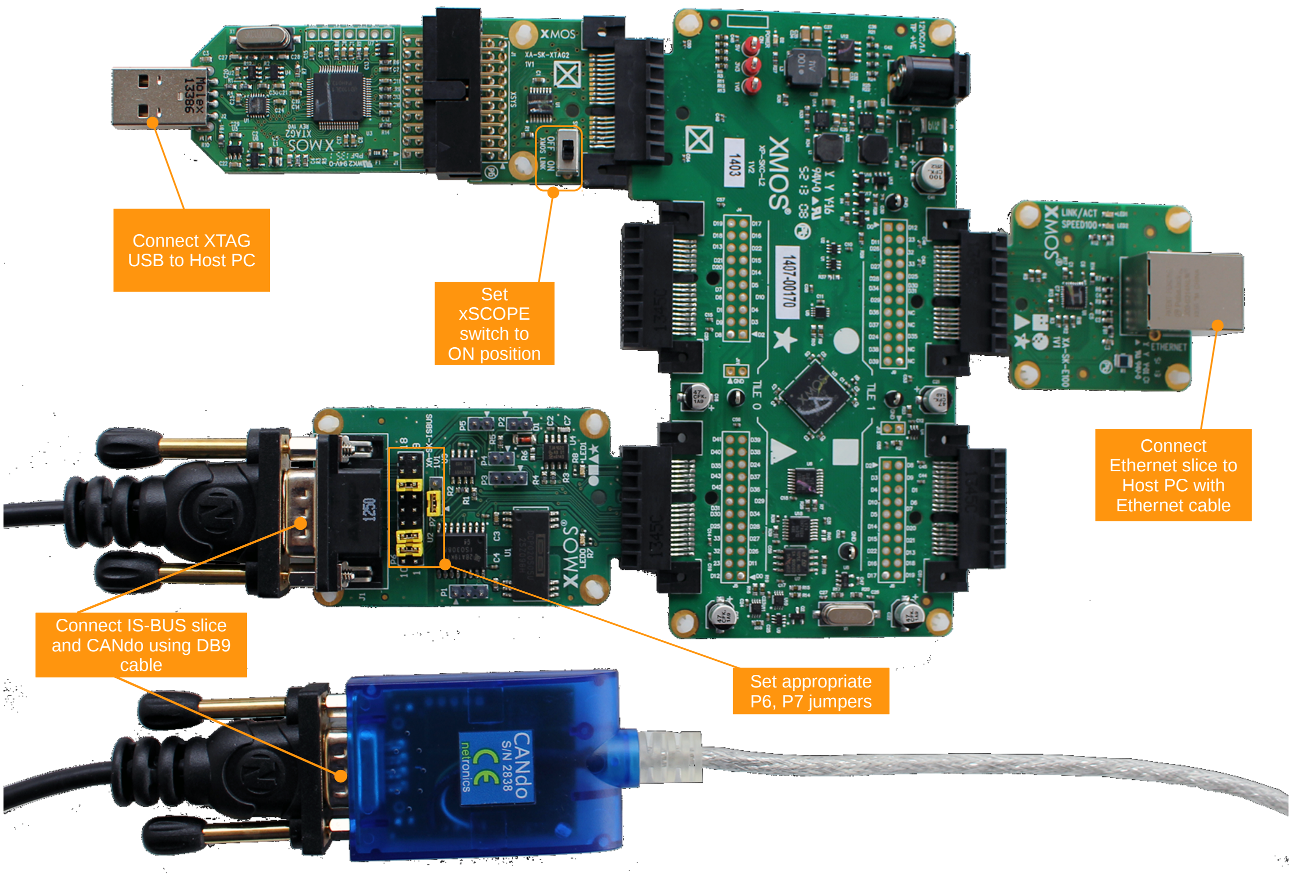

Connect XTAG USB to Host PC
Set xSCOPE switch to ON position
Connect Ethernet slice to Host PC with Ethernet cable
Connect IS-BUS slice and CANdo using DB9 cable
Set appropriate P6, P7 jumpers

Figure 3: XMOS xCORE-L16 sliceKIT set up for CAN-Ethernet bridge

To run the demo,

1. Connect IS-BUS slice (XA-SK-ISBUS) to the sliceKIT core board using the connector marked with the TRIANGLE
2. Set the jumpers on the IS-BUS slice for CAN mode; P7 short between pins 1 and 2 (leaving 3 unconnected), P6 short between 2 and 11, 3 and 12, 7 and 16
3. Connect the IS-BUS slice to the CANdo USB interface via a DB-9 pass through cable
4. Connect Ethernet slice (XA-SK-E100) to the sliceKIT core board using the connector marked with the CIRCLE
5. Connect one end of the Ethernet cable to Ethernet slice and the other end to the RJ45 jack of your host PC
6. Connect xTAG-2 debug adapter

7. Connect the xTAG-2 to your development PC
8. Set the `XMOS LINK` to `ON` on xTAG-2 debug adapter
9. Switch on the power supply to the sliceKIT core board

Note: The example setup provided here uses a Windows machine but it can be executed on a Linux host. You must install the drivers and UI application required for the appropriate CAN to USB dongle before trying this demo on your Linux machine.

A.1 Windows Host

1. Install the CANdo application on your Windows machine and connect the CANdo dongle to your PC
2. Within the CANdo application:
   (a) Click the `CAN Setup` tab and set the baud rate to *1M*.
   (b) Click `View -> Options` then ensure the `Display On CAN View Page` option is checked. Then click `OK`.
   (c) Switch to the `CAN View` tab.
   (d) Click the green (Start CANdo) button
3. Install python (version 2.6 or later) on your Windows machine

APPENDIX B - Launching the demo device

Once the demo example has been built either from the command line using xmake or via the build mechanism of xTIMEcomposer studio you can execute the application on the sliceKIT core board.

Once built there will be a `bin` directory within the project which contains the binary for the xCORE device. The xCORE binary has a XMOS standard .xe extension.

B.1 Launching from the command line

From the command line, use the xrun tool to download code to the xCORE device. Navigate to the bin directory of the project and execute the code on the xCORE microcontroller as follows:

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

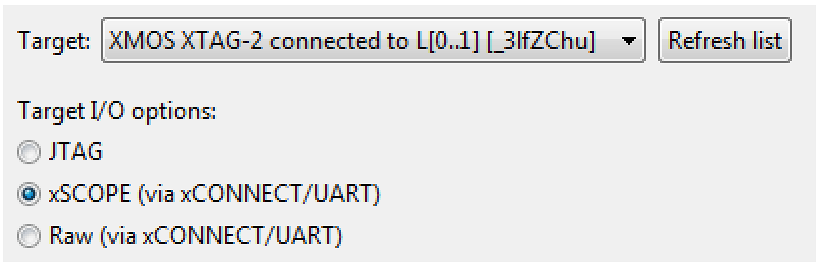
Once this command is executed, you will see the following text in the console window:

```
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 Launching from xTIMEcomposer Studio

From xTIMEcomposer Studio, use the run mechanism to download code to xCORE device. Select the xCORE binary from the bin directory, right click and then follow the instructions below:

- Select **Run Configuration**.
- Enable xSCOPE in Target I/O options:


Target: XMOS XTAG-2 connected to L[0..1] [_3lfZChu]

Target I/O options:
JTAG [✓ xSCOPE (via xCONNECT/UART)] Raw (via xCONNECT/UART)

Figure 4: xTIMEcomposer xSCOPE configuration

- Click **Apply** and then **Run**.

When the processor has finished booting you will see the following text in the xTIMEcomposer console window:

```
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
```

APPENDIX C - Running the demo

C.1 Windows Host

- Select the CANdo `application` and ensure it successfully lists your CANdo interface
- Open a console window and execute a host test script (a TCP socket client) as:

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

Note: Ensure you provide the correct source location of your test script and provide the IP address of your XMOS device as displayed in the xTIMEcomposer console window.

- Once the host test script is able to connect to your XMOS device, it displays the following:

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

- The host test script will start sending the data packets (5 x 8-byte CAN data packets as test data) to the XMOS device as follows:

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

- Select CAN `View` tab on the CANdo `application`. Check if it displays any *Rx* frames which contain the data sent from the host test script. If it does not display, double check if the connections and setup is proper, and try the demo steps again
- Select the same data to be sent from the CANdo. The XMOS device receives this CAN frame and it is sent to the host script.
- The host script receives this CAN frame and displays on its console:

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

APPENDIX D - References

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

APPENDIX E - Full source code listing

E.1 Source 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;

         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 Source code for 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;
}
```