life.augmented

AN5517

Application note

A gateway implementation using the SPC58ECxx devices

Introduction

An emerging trend in the automotive industry is the use of an advanced gateway that enables secure and reliable communications among heterogeneous vehicle networks.

There is a multitude of in-vehicle bus systems, for instance LIN, CAN, Ethernet, which are all used for connecting different domains like body, powertrain, chassis, safety, infotainment and driver assist in a vehicle.

A gateway may have various levels of complexity according to the protocols and message encapsulation policies, security features and performance requirements.

STMicroelectronics SPC58x Microcontroller family offers platforms designed for this purpose embedding specialized devices which can meet the requirements of a full-feature gateway.

This application note shows a gateway implementation by using the SPC584Cx/SPC58ECx 32-bit MCU, which is part of a family of microcontrollers that targets automotive vehicle body and gateway applications.

The application is written using the SPC5Studio tool which is available on https://www.st.com/en/development-tools/spc5-studio.html.

The SPC5Studio provides an advanced software driver and routing algorithm plus a user-friendly interface (with graphical wizard) which allows to configure and set up the gateway.

The application note uses a demo software, available in the SPC5Studio, that can be mainly used to demonstrate: the routing among all the involved protocols and how different messages are encapsulated to be transferred among the peripherals.

No security features and performance figures are provided in the application note, since this is mainly focused on features and routing strategy.

1 Gateway overview

A gateway manages data messages among different protocols.

Communication protocols have different message transmission speeds, communication paradigms and message frames.

The application covers several scenarios:

- Ethernet to CAN;
- CAN to Ethernet;
- CAN to CAN;
- CAN to Lin (master);
- Ethernet to Lin (master);
- Lin (slave) to CAN;
- Lin (slave) to Ethernet;
- Lin Master / Slave.

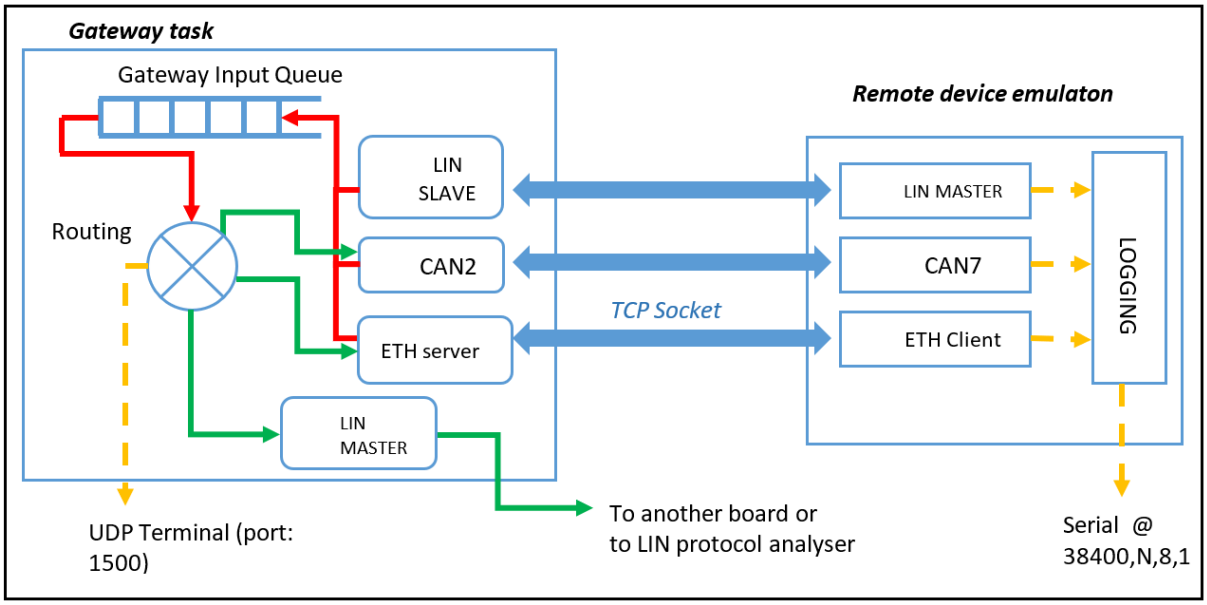
The application software runs on a single core that is designed to implement a gateway.

To stimulate the gateway, having a self-contained application, the operating system schedules a task to start transferring messages among the different protocols. This task runs on the same CPU core. It is called remote device emulation task.

Removing the remote device, the gateway can be stimulated by using external hardware connections.

Figure 1. Gateway block schema shows the entire application design model that is composed by a gateway task and a remote device emulation task.

Figure 1. Gateway block schema



The entire application has the advantage of working on top of the FreeRTOS operating system with the TCP/IP stack (available in the SPC5Studio suite).

The operating system and network stack are tuned so no further changes in the configuration are needed.

In a glance, from the Figure 1. Gateway block schema, the gateway reads the messages from the receiving queue and dispatches them by using a dedicated routing table. In the demonstration the messages routed to the Ethernet controller are not accumulated to fill an entire Ethernet frame.

The remote devices log actions (message sent/received) by using the serial port.

A serial port is available via the USB debug connector; its configuration is 38400,N81. The gateway logs actions through a UDP data stream. More details about the serial port and UDP logging can be found in the Appendices at the end of this document.

2 Ethernet controller

The SPC58x automotive microcontrollers feature a Quality-Of-Service 10/100 Mbit/s Ethernet controller that implements the Medium Access Control (MAC) Layer plus several internal modules for standard and advanced network features.

Main features can be divided into the following categories:

- MAC core
- MAC Transaction Layer (MTL)
- DMA block
- SMA interface
- MAC management counters
- Power Management block (PMT)

The MAC core implements both main transmission and reception features and advanced standards. The latter are sometimes optional modules for example Power Management Module, Energy Efficient Ethernet and so on. These are configurable features that can be present only on some configurations.

The Transaction layer (MTL) block consists of transmit and reception FIFOs that can be also configurable with different sizes and optimizations.

The Direct Memory Access is an advanced DMA block used to exchange data between the internal FIFOs and the system memory. The engine provides an interrupt to control transmit and receive packets and other options for routing packets to different channels based on VLAN tagging.

The Ethernet can access the PHY registers through the Station Management Agent (SMA) module. The PMT is designed to manage the low power and Wake Up features (e.g. Magic packet).

The MAC Management Counters (MMC) module allows to enable various counters according to the Remote Monitoring (RMON) standard specification (RFC2819/RFC2665).

3 CAN controller

The SPC584Cx/SPC58ECx has eight CAN instances embedded in two different subsystems as documented in the device reference manual.

All CAN controllers present in the same subsystem will share resources like RAM memory, clock, etc.

Each CAN subsystem consists of the following major blocks:

- Modular CAN cores: The registers of the CAN module can be accessed using the Generic Slave Interface (GSI). The peripheral GSI module enable acts as a request from each master.
- CAN-RAM arbiter: is an additional logic for arbitration between the requests for the RAM access by the various CAN controllers.
- SRAM: The CAN subsystem will interface with an external RAM using this interface.
- ECC Controller: it contains the logic to compute and validate the correction code on the SRAM memory.

4 LinFlexD controller

The SPC584Cx/SPC58ECx embeds up to 18 LINFlexD controllers which are designed to manage efficiently a high number of LIN messages with a minimum of CPU load offering the possibility the possibility to use dedicated DMA channels.

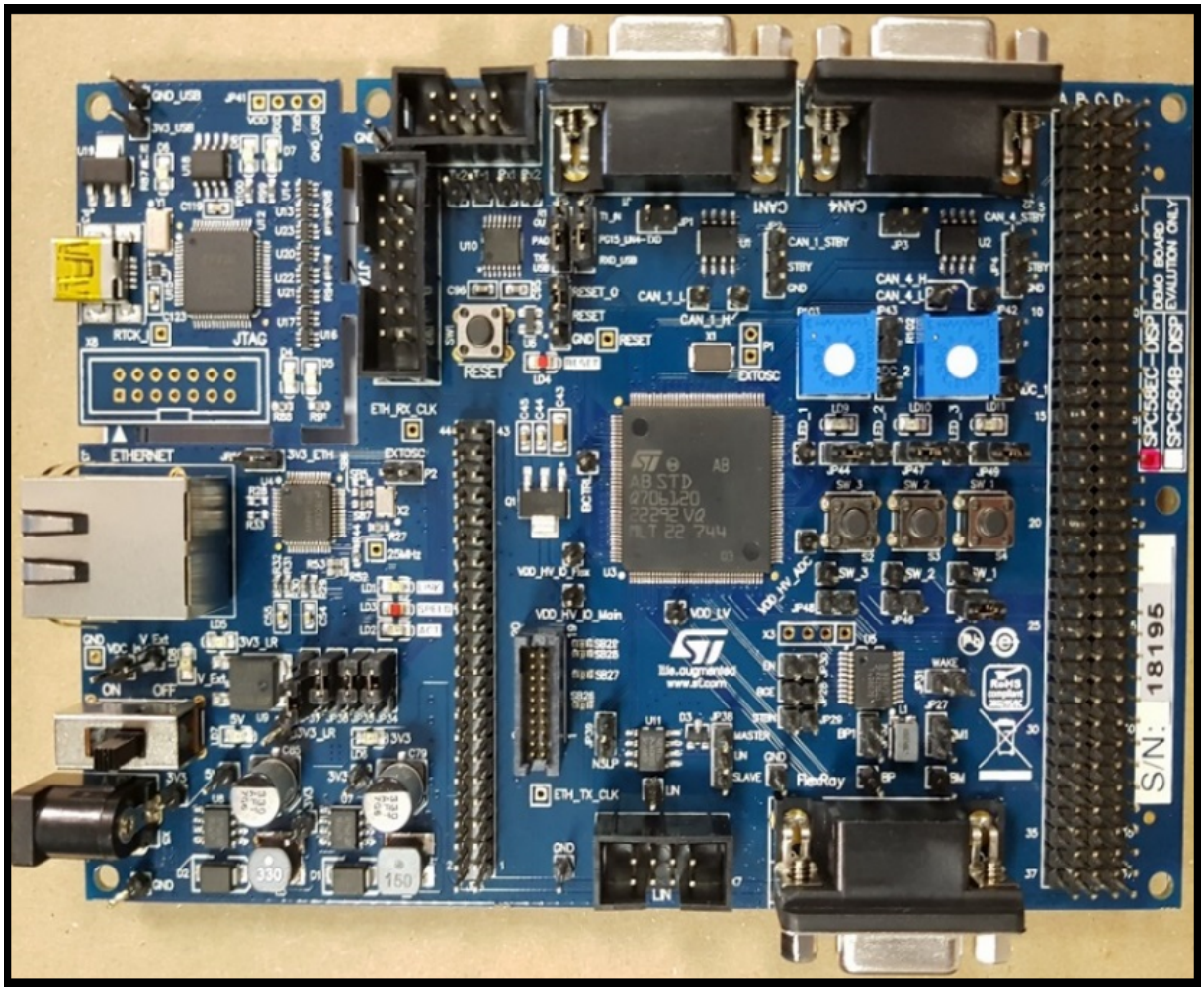
The LINFlexD supports LIN protocol version 1.3, 2.0, 2.1 and 2.2 with Bit rates up to 20 Kbit/s (LIN protocol).

The LINFlexD also provides support for some of the basic UART transfers which are able to achieve 25Mbits/s.

5 Hardware setup

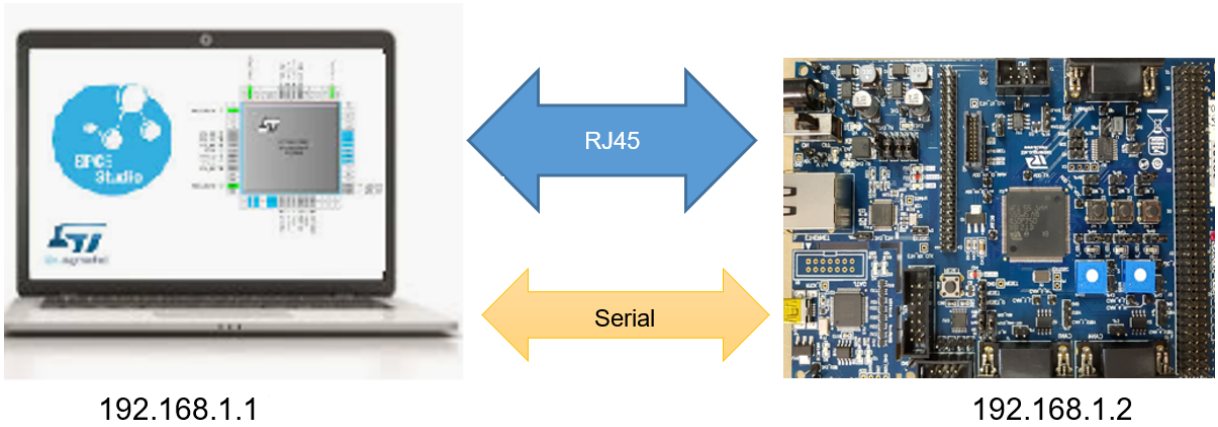
The reference platform used for this gateway implementation is the SPC58EC-DISP which hosts a 4MB Flash memory 32-bit SPC58 automotive microcontroller.

Figure 2. SPC58EC-DISP



The following figure shows a simple setup used for networking.

Figure 3. **Network setup**



The SUBSYS_0_CAN_1 and SUBSYS_1_CAN_4 need to be connected: High to High and Low to Low CAN lines (or wiring the external D-SUB 9 WAY FEMALE connectors).

LinFlex master/slave communication is implemented so the two instanced LIN0 and LIN12 must be connected together crossing RX and TX signals on the PCB:

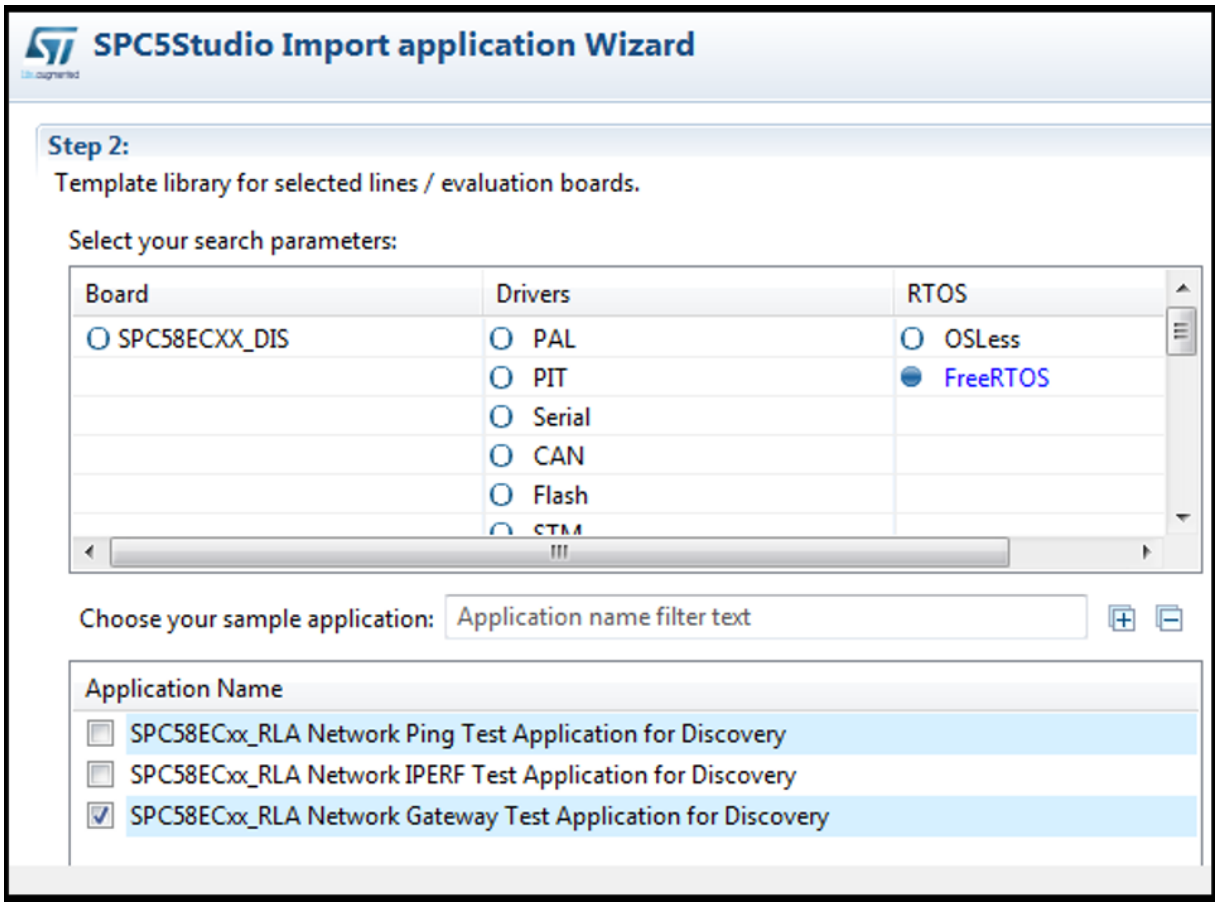
- LIN0_TXD (D1) wired to LIN12_RXD (B22)
- LIN0_RXD (A16) wired to LIN12_TXD (A25)

The Jumper set (JP_5) must be all closed to make the Ethernet work; the JP_50 has to be closed, too

6 Peripheral configurations

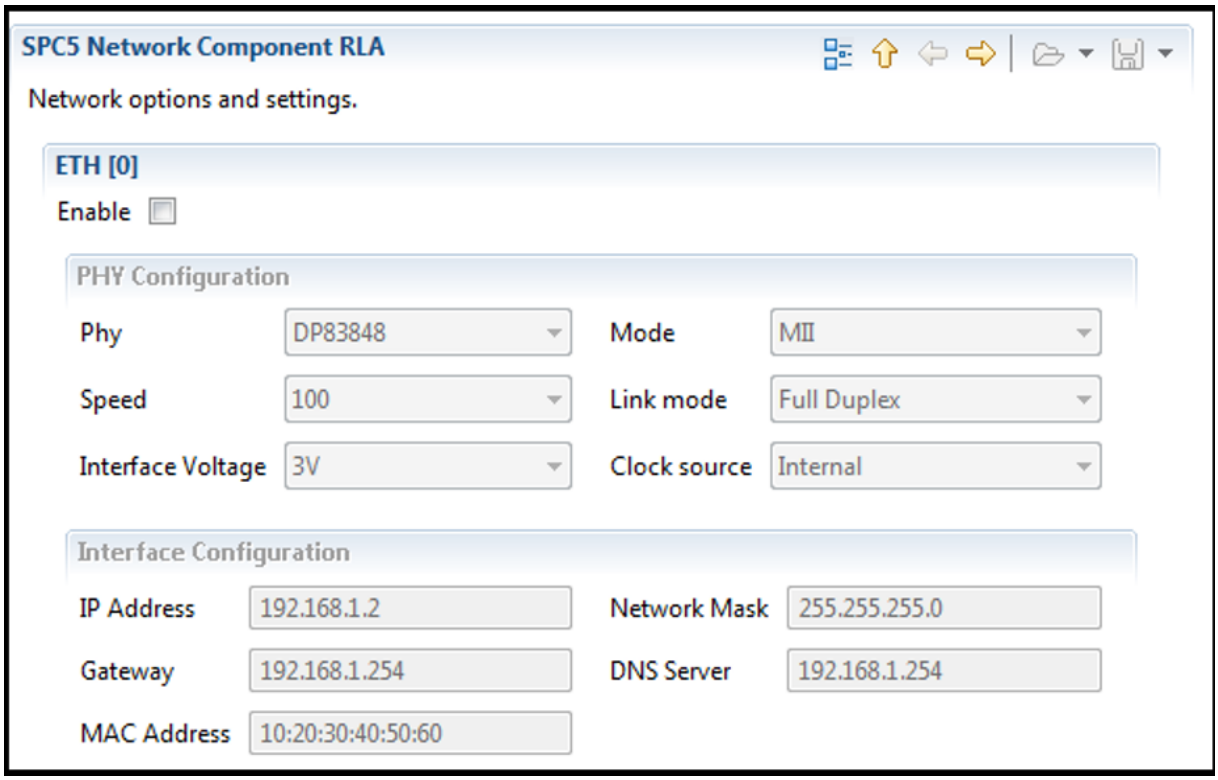
The following figure shows how to select the gateway application designed for the SPC58EC-DISP board.

Figure 4. SPC58ECxx OS network demos



The SPC5Studio network component is used to configure both Ethernet and PHY devices. The graphical interface offers several options to configure the devices. Default configuration guarantees the full stack working on this platform.

Figure 5. **SPC5Studio network component**



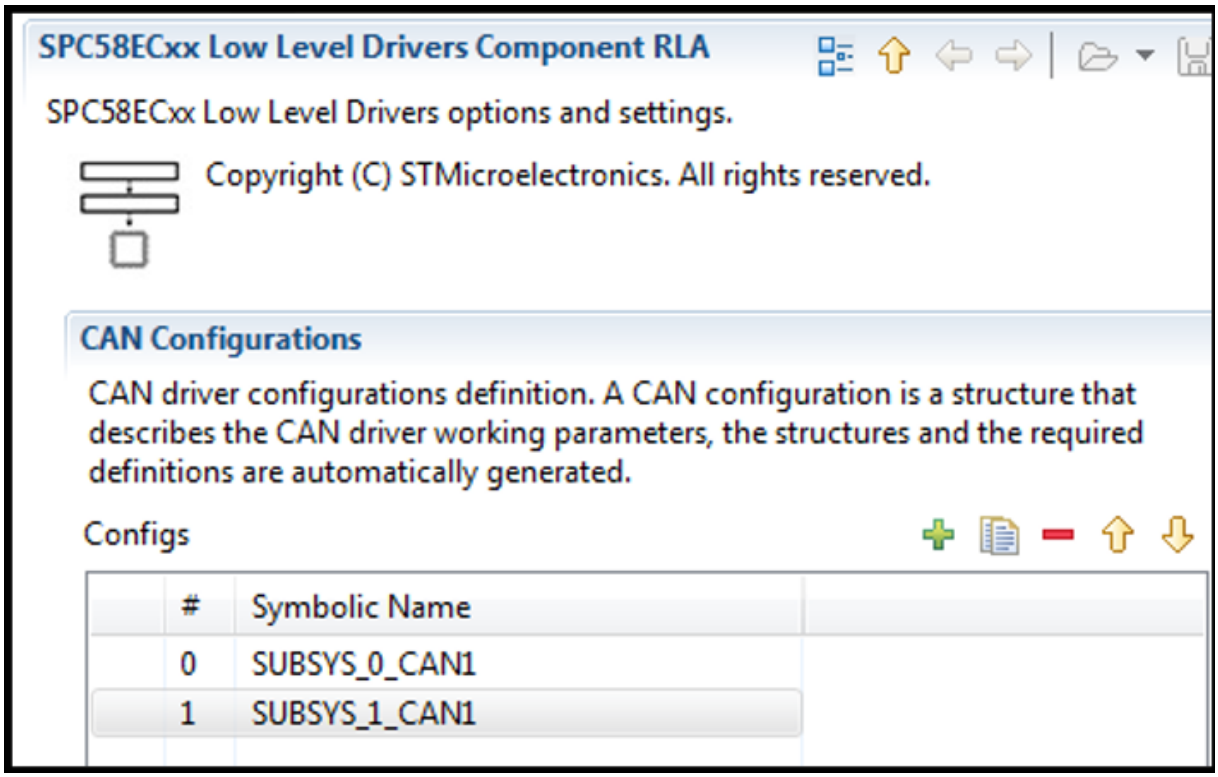
Two instances of the CAN are configured with standard filters, 8MHz of protocol clock and 1M as bit timing. No loopback mode is configured, in fact the two instances are externally wired.

On each instance three RX filters are applied by default. For example, for SUBSYS_0_CAN1 only the 0x7d0, 0x7f8, 0x7f7 message IDs are passed to the controller. While the following IDs: 0x7d0, 0x7e0, 0x7f0.

Further details about the CAN bus configuration can be found in AN5416 SPC58EC CAN bus configuration,

The default configuration can be changed by using the related graphical interface in SPC5Studio:

Figure 6. **SPC5Studio CAN configuration**



The following table explains the name conventions used for the CAN instances in the project:

Table 1. **CAN instance naming**

CAN instance in a subsystem	I/O pin table	SPC5Studio Configuration	SPC5Studio CAN_DEVICE	PCB
CAN_SUB_0_M_CAN_1	CAN 1 sys0	SUBSYS_0_CAN1	CAN2	CAN1
CAN_SUB_1_M_CAN_1	CAN 4 sys0	SUBSYS_1_CAN1	CAN7	CAN4

Figure 7. SPC5Studio LinFlex configuration shows the low level driver component to configure the three Lin instances used in the project.

Figure 7. **SPC5Studio LinFlex configuration**

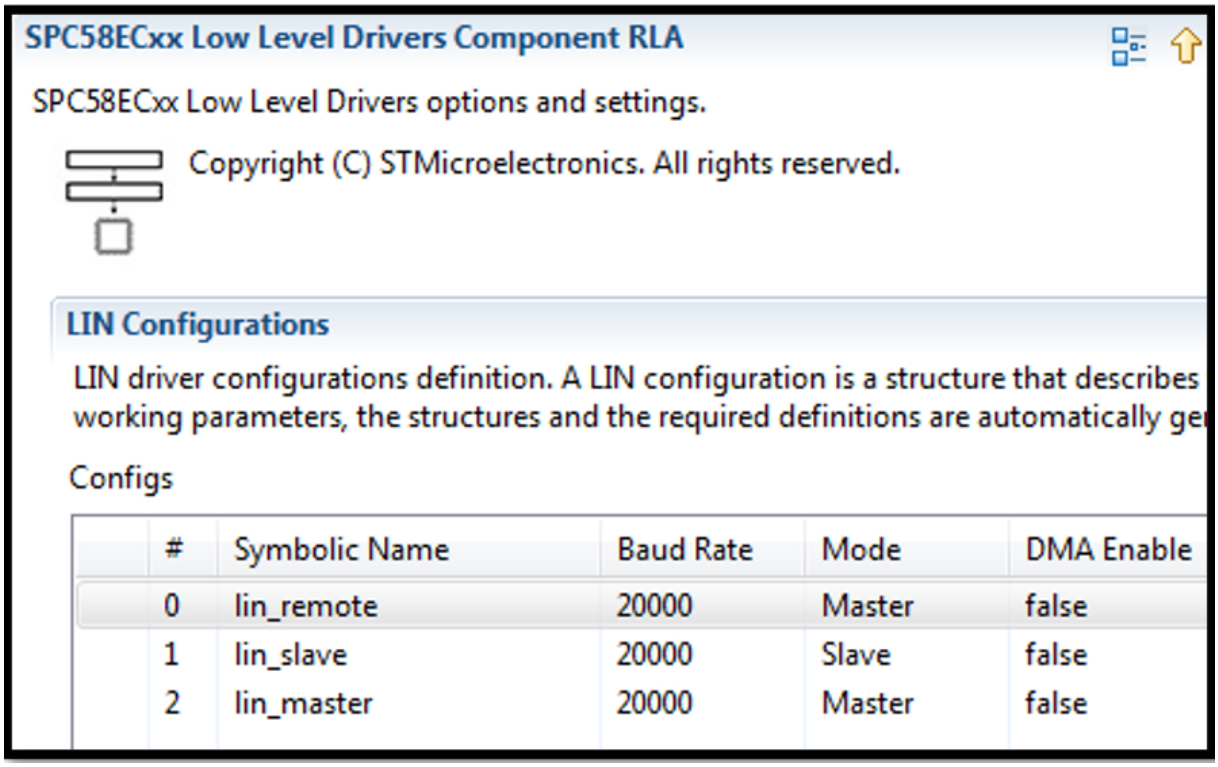


Table 2. **LIN instance naming**

SPC5Studio Configuration	SPC5Studio Lin Instance	Peripheral
LIN_REMOTE	LD3	LinFlex12
LIN_SLAVE	LD1	LinFlex0
LIN_MASTER	LD1	LinFlex6

Note that the LinFLEX instance 4 is configured in UART mode and used in the remote device part to dump the output.

By default the LinFlex master instances are configured with synchronous mode, and the read and write operations will be blocked until finished. The baudrate is 20Kmbps. The DMA can be enabled by using the related interface.

The LinFlex slave also uses the same baudrate and no DMA in the demo application. Two callbacks are used to manage receive and transmit events (lin_slave_tx_callback/ lin_slave_rx_callback) and, in slave mode, dedicated filters are associated for some id.

7 Scheduling the tasks

The main function creates the necessary tasks before starting the OS and the entire network stack.

These are the two main tasks used to implement the two main tasks to implement the gateway and the remote device emulator.

```
ret = xTaskCreate(remoteDevTask, "remoteDevTask",
REMOTE_DEVICE_TASK_STACK_SIZE, NULL,
REMOTE_DEVICE_TASK_PRIORITY, NULL);
```

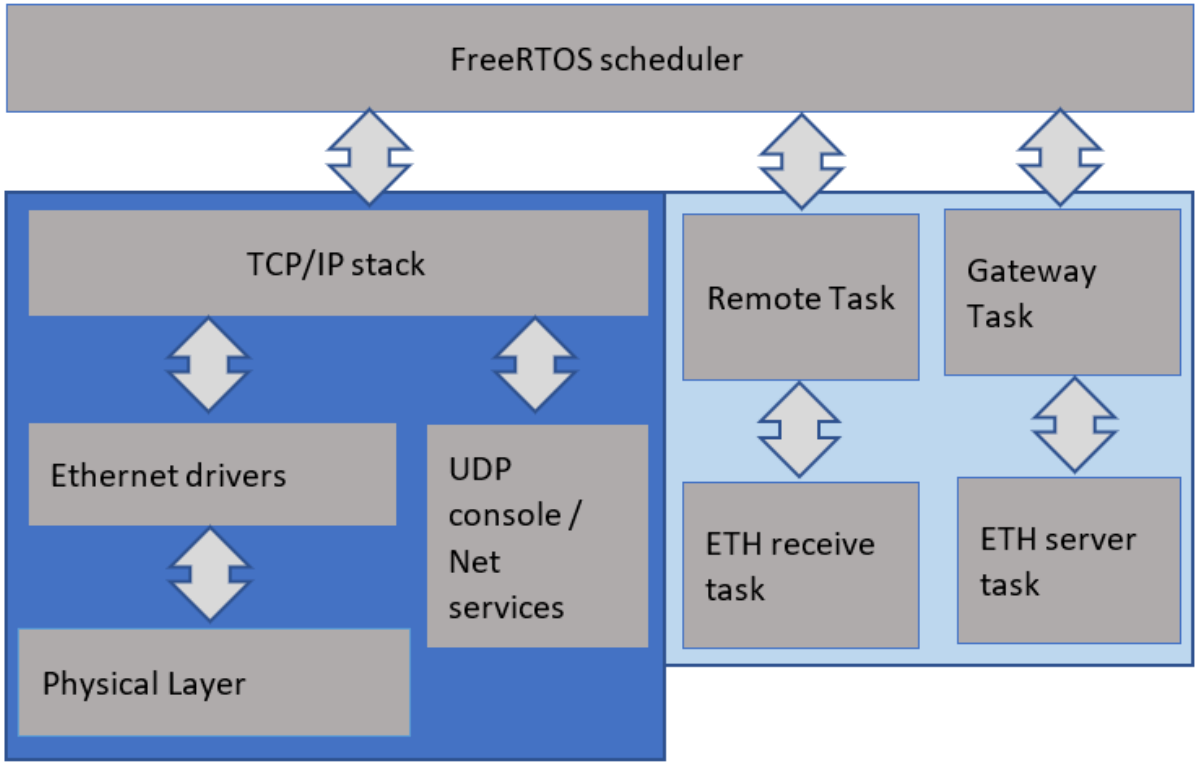
```
ret =
        xTaskCreate(remoteDevTask, "remoteDevTask", REMOTE_DEVICE_TASK_STACK_SIZE, NULL,
        REMOTE_DEVICE_TASK_PRIORITY, NULL);
```

The initialization of the entire network is done through the `networkInit()` function, which is called by the `componentInit()` helper.

The FreeRTOS_IPInit() is responsible for inizializing the FreeRTOS+TCPIP stack.

The figure below shows a simplified flow diagram of the tasks that are scheduled by the operating system when starting the application:

Figure 8. FreeRTOS tasks



The remote and gateway tasks will be detailed in the next paragraphs.

7.1 Remote device emulator task

The *remoteDevTask* performs the following steps (refer to the Figure 8. FreeRTOS tasks):

- Initialize the serial console.
- Initialize LIN remote master:
  - by invoking the `lin_lld_start()` low level LinFLEX API in SPC5Studio.
- Initialize CAN7 instance:
  - by invoking the `can_lld_start()` low level CAN API in SPC5Studio.
- Start the Ethernet client:
  - create a Socket by using the `FreeRTOS_socket()` API.
  - let the gateway task to start the Ethernet server.
  - connect a TCP socket to the gateway socket (`FreeRTOS_connect()`).
  - create a new task to receive (`FreeRTOS_recv()`).
- Start a random message traffic generation.
  - In a loop different messages with different IDs are transmitted over the configured peripherals: CAN, LinFlex and Ethernet as shown in the example code below:

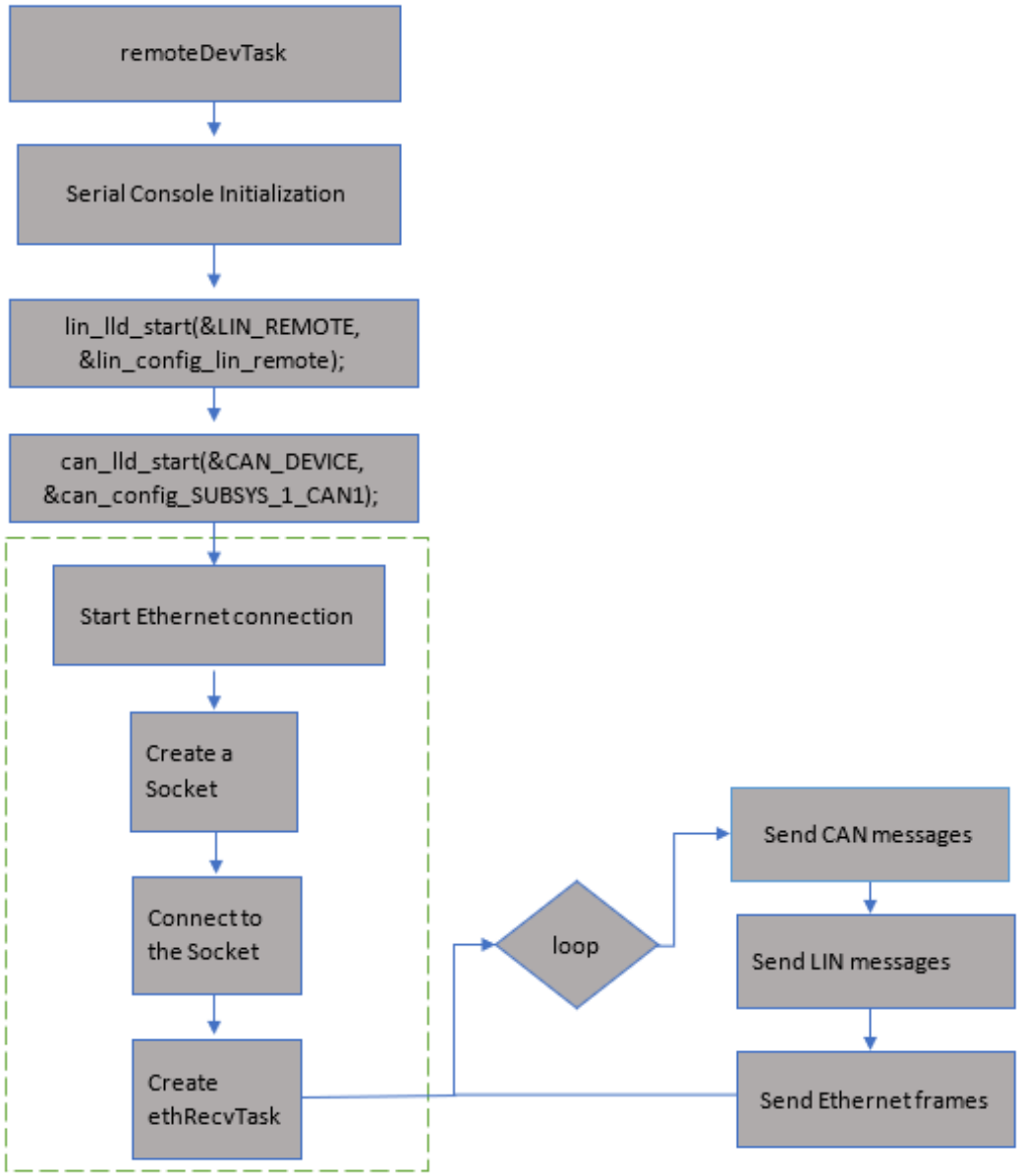
```
/* Send LIN_MSG_2 from LIN_REMOTE master */
lin_lld_transmit(&LIN_REMOTE, LIN_MSG_2, lin_tx_buf, LIN_MSG_2_SIZE);
/* Send CAN_MSG_1 */
can_data[0] = counter;
can_data[1] = 0x11223344;
can_send_message(CAN_MSG_1, can_data[0], can_data[1]);
LOG_CAN_MSG(0, 'T', CAN_MSG_1, can_data, 8); /* To the serial console */
/* Send ETH_MSG_2 */
eth_tx_buf[0] = ETH_MSG_2;
for (i = 1; i <= ETH_DATA_LEN; i++) {
      eth_tx_buf[i] = (uint8_t)'A' + i;
}
eth_send_message(eth_tx_buf, ETH_DATA_LEN + ETH_HEADER_LEN);
LOG_ETH_MSG(0, (uint8_t)'T', ETH_MSG_2, eth_tx_buf, 8);
```

The `eth_send_message()` function invokes the following API to send a frame to the gateway (refer to the next sections): FreeRTOS_send(gateway_socket, &data[p], len, 0);

While the receive task will receive data from the gateway socket, as shown below:

```
portTASK_FUNCTION(ethRecvTask, pvParam ) {AN5517 - Rev 1 page 14/32
…
for( ;; ) {
    n = FreeRTOS_recv(gateway_socket, buf, 16, 0);
    if (n > 0) {
            LOG_ETH_MSG(0, (uint8_t)'R',
            buf[0], &buf[1], n - 1);
…
```

Figure 9. **Remote device block schema**



7.2 Routing tables

Routing tables describe how messages are transferred across different peripherals. A routing table contains the following fields:

```
typedef struct RoutingTable_s
{
   uint32_t src_msg_id; /* Source message */
   uint32_t action; /* How to route the message */
   void *dst_device; /* Destination device */
   uint32_t dst_msg_id; /* Destination message id */
} RoutingTable_t;
```

Below the table adopted for each protocol.

```
/* LIN routing table */
RoutingTable_t lin_routing_table[] = {
    { LIN_MSG_2, LIN_TO_CAN, &CAN_DEVICE, CAN_MSG_2},
    { LIN_MSG_3, LIN_TO_LIN, &LIN_MASTER, LIN_MSG_3},
    { LIN_MSG_4, LIN_TO_ETH, &ETH_DEVICE, ETH_MSG_3},
    { 0, INVALID_ACTION, NULL, 0}
};
```

For example, the CAN_MSG1 (so the ID = 0x7d0) is always routed from CAN7 to CAN2 with CAN_MSG3 (ID=0x7f0).

```
/* CAN routing table */
RoutingTable_t can_routing_table[] = {
    { CAN_MSG_1, CAN_TO_CAN, &CAN_DEVICE, CAN_MSG_3},
    { CAN_MSG_4, CAN_TO_LIN, &LIN_MASTER, LIN_MSG_3},
    { CAN_MSG_5, CAN_TO_ETH, &ETH_DEVICE, ETH_MSG_1},
    { 0, INVALID_ACTION, NULL, 0}
};
/* ETH routing table */
RoutingTable_t eth_routing_table[] = {
    { ETH_MSG_1, ETH_TO_CAN, &CAN_DEVICE, CAN_MSG_3},
    { ETH_MSG_2, ETH_TO_LIN, &LIN_MASTER, LIN_MSG_3},AN5517 - Rev 1 page 16/32
    { ETH_MSG_3, ETH_TO_ETH, &ETH_DEVICE, ETH_MSG_4},
    { 0, INVALID_ACTION, NULL, 0}
};
```

The gateway will use the following table to route the messages among the protocols adopting the rules defined in the table above:

```
/* Routing tables */
static RoutingTables_t routing_tables[] = {
    { GATEWAY_MSG_LIN, lin_routing_table },
    { GATEWAY_MSG_CAN, can_routing_table },
    { GATEWAY_MSG_ETH, eth_routing_table }
};
```

The *routing,c* file exposes the following API used by the gateway:

- `get_routing_table()`:
  - to route the message according to the table
- `get_action()`:
  - how to route the message
- `get_destination_device()`
- `get_destination_message_id()`

7.3 Starting the gateway

The gateway, before creating its own task, prepares a dedicate queue: the `gatewayQueue` is the returned handler that will be used by each protocol:

```
gatewayQueue = xQueueCreate(GATEWAY_QUEUE_SIZE, GATEWAY_QUEUE_ITEM_SIZE);
```

Note that the operating system is able to dynamically allocate the memory, this is required to support queues.

The gatewayTask performs the following steps:

- Start the LinFlex slave;
- Start the CAN2 instance;
- Start the Ethernet server;
  - Open a new listener socket;
  - Bind to the socket;
  - Waiting for the connection and start the server task, it will
    - receive the packets;
    - add messages to the queue;
    - notify the gateway task that there is pending job;
- The gatewayTask will wait for a new message to route, on this event, it will:
  - Get the message from the queue
  - Get routing table for the incoming message type (LIN, CAN, ETH) and message id
  - Route the data (invoking send_to_can, send_to_lin, send_to_eth to call the low level API and transfer the message to the device)

The two functions below are invoked by the CAN and Lin (slave) ISR to add the message in the queue and notify the event to the gateway task.

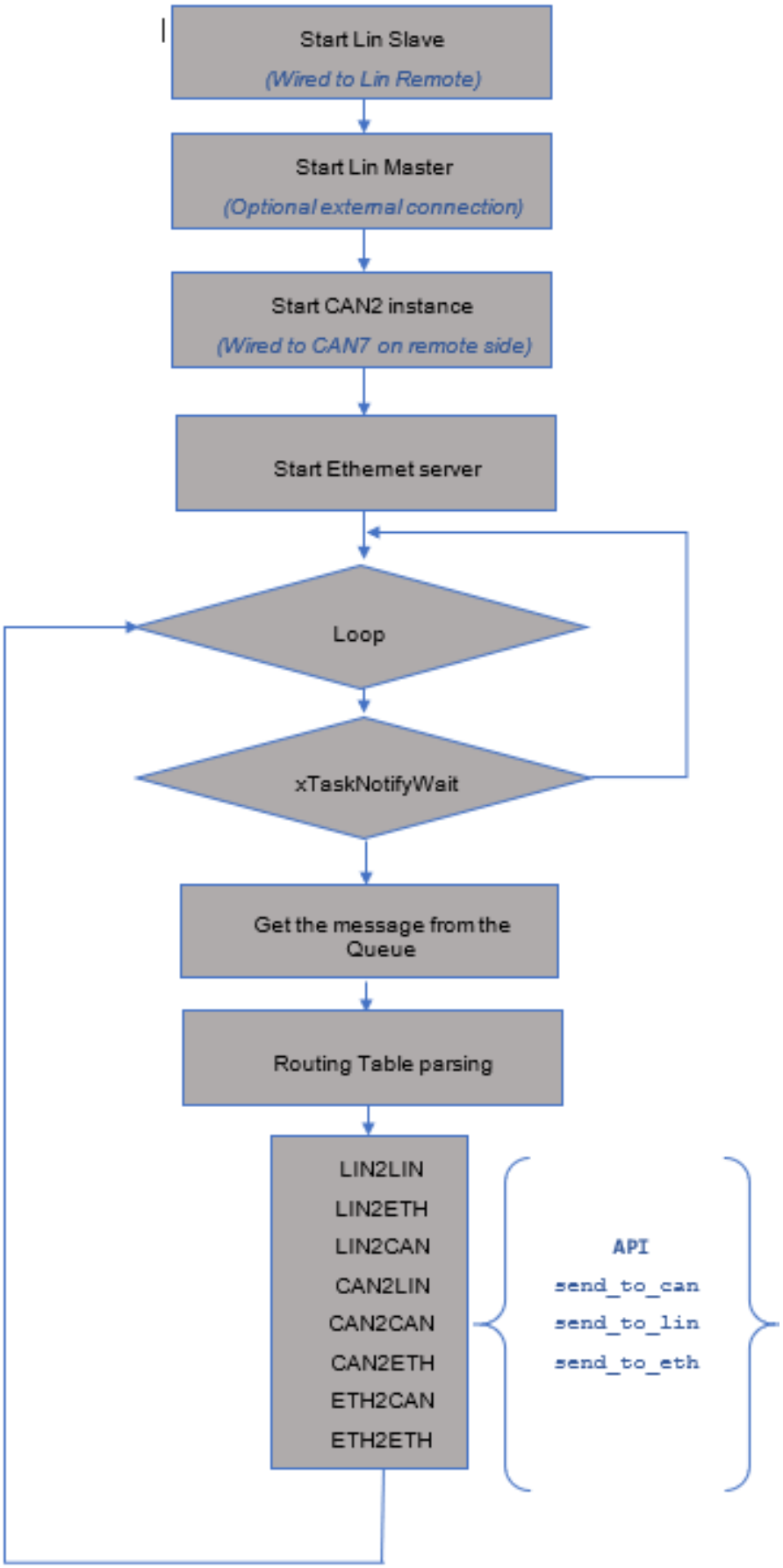
```
void send_to_gateway_from_ISR(GatewayMsg_t *gw_msg) {
    /* Add message to gateway data queue. */
    xQueueSendToBackFromISR(gatewayQueue, gw_msg, 0);v 1 page 17/32
    /* Notify gateway task of a new message. */
    xTaskNotifyFromISR(gatewayTaskHandle, 0, eSetBits, NULL);
}
```

The function below is called by the Ethernet server task.

```
void send_to_gateway_from_task(GatewayMsg_t *gw_msg) {
    taskENTER_CRITICAL();
    /* Add message to gateway data queue. */
    xQueueSendToBack(gatewayQueue, gw_msg, 0);
    /* Notify gateway task of a new message. */
    xTaskNotify(gatewayTaskHandle, 0, eSetBits);
    taskEXIT_CRITICAL();
}
```

Figure 10. Gateway task block schema shows the flow diagram of the operstions performed by the gateway task.

Figure 10. **Gateway task block schema**



8 Routing examples

The examples reported in this section describe the flow of the routed messages among the different protocols.

8.1 LinFlex to Ethernet and CAN to Ethernet

Considering the case where the remote task invokes the following function to send a message from the LIN_REMOTE master:

```
/* Send LIN_MSG_4 */
lin_lld_transmit(&LIN_REMOTE, LIN_MSG_4, lin_tx_buf, LIN_MSG_4_SIZE);
```

The slave will receive the message and the LinFlex interrupt service routines invokes:

```
lin_slave_rx_callback(LinDriver *ldp, uint8_t idMessage, uint8_t *buffer, uint16_t len) {
…
    gw_msg.type = GATEWAY_MSG_LIN;
    gw_msg.id = idMessage;
    memcpy(gw_msg.data, buffer, len);
    gw_msg.data_len = len;
    send_to_gateway_from_ISR(&gw_msg);
}
…
```

As shown in the previous chapter the `send_to_gateway_from_ISR()` will notify the gateway that there is a new element in the queue. So the gateway task will have to manage it.

In details the gateway task will find that, from the routing table, this message has to be routed to the Ethernet controller.

```
RoutingTable_t lin_routing_table[] = {
    { LIN_MSG_2, LIN_TO_CAN, &CAN_DEVICE, CAN_MSG_2},
    { LIN_MSG_3, LIN_TO_LIN, &LIN_MASTER, LIN_MSG_3},
    { LIN_MSG_4, LIN_TO_ETH, &ETH_DEVICE, ETH_MSG_3},
…
```

So the gateway task will enter in this case:

```
case
LIN_TO_ETH:N5517 - Rev 1 page 20/32
    send_to_eth(device, dst, gw_msg.data, gw_msg.data_len);
    LOG_MESSAGE("LIN TO ETH",    gw_msg.id, dst, gw_msg.data_len, gw_msg.data);
    break;
```

When invoking the `send_to_eth()` function, the data is copied in the packet data and the `FreeRTOS_send()` is called to send on `eth_socket` socket. Actually the packet will be moved by the network stack to the Ethernet controller. It will copy the data into its own DMA ring and send it.

The LOG_MESSAGE will invoke the `FreeRTOS_printf()` which in this project dumps the data through the Ethernet as UDP datagrams (see Section Appendix A UDP Logging).

Similar logic can be applied in the case when the CAN message is passed to the Ethernet instead of the LinFlex. In this case, the remote will send the CAN_MSG_5 from the CAN7. The CAN2 will receive the message and notify the gateway by calling the `send_to_gateway_from_ISR()` that there is a new entry in the queue. The gateway task will decode the message understanding it must be sent by the network stack as done for the Lin case above.

8.2 CAN to CAN

Considering the case when, on the remote device, the following message is sent:

```
can_data[0] = counter;
can_data[1] = 0x11223344;
can_send_message(CAN_MSG_1, can_data[0], can_data[1]);
```

The CAN7 will send a message that will be received by the CAN2 and its interrupt service routine will invoke the associated callback:

```
void SUBSYS_0_CAN1_rx_callback(uint32_t msg_buf, CANRxFrame can_msg) {
…
    gw_msg.type = GATEWAY_MSG_CAN;
    gw_msg.id = can_msg.ID;
    memcpy(gw_msg.data, can_msg.data32, can_msg.DLC);
    gw_msg.data_len = can_msg.DLC;
    send_to_gateway_from_ISR(&gw_msg);
…
```

It will notify the gateway task to route this message to the CAN:

```
RoutingTable_t can_routing_table[] = {
…
    { CAN_MSG_1, CAN_TO_CAN, &CAN_DEVICE, CAN_MSG_3},
…
```

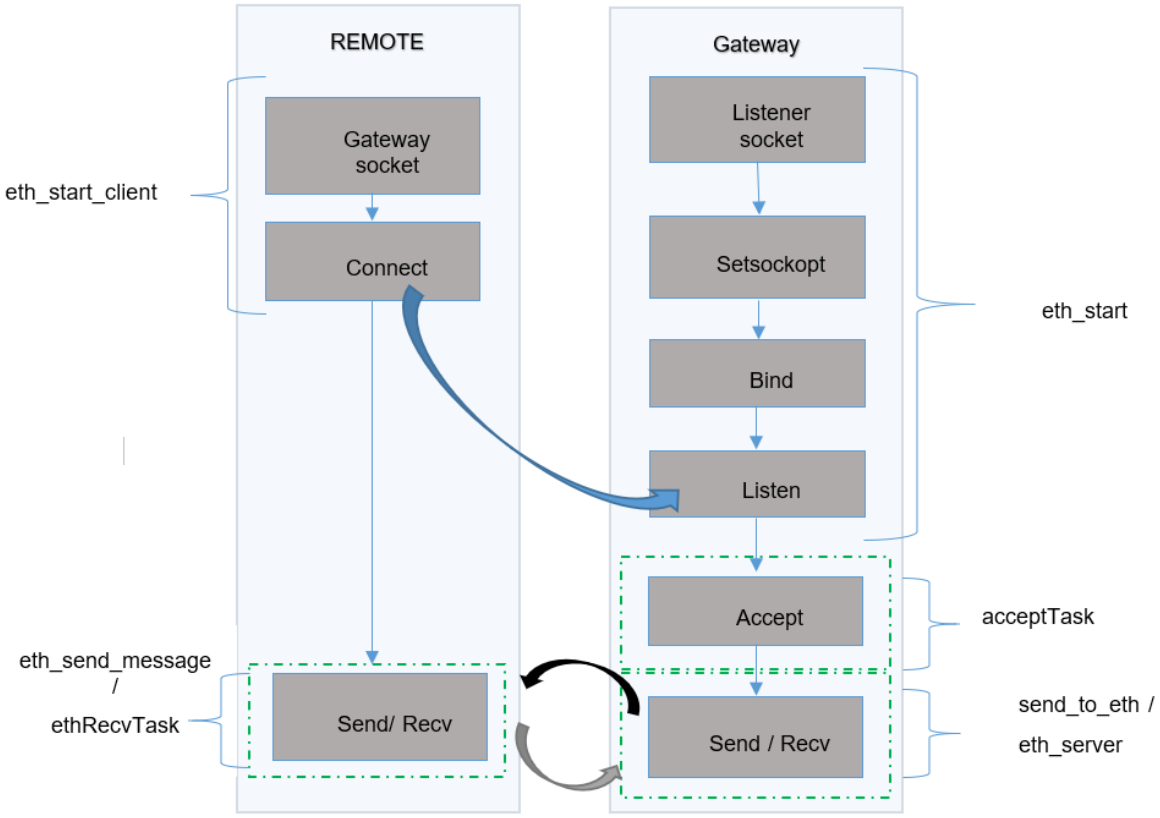
The received message has the ID 0x7d0, please remember that this matches the filter setup for the SUBSYS_0_CAN1 (i.e.: 0x7d0, 0x7f8, 0x7f7).

The routed message will have the new 0x7f0. Note that the SUBSYS_1_CAN1 will receive this message because it matches the filter settings.

8.3 Ethernet to Ethernet

Let's consider the case that packets are moved between the single Ethernet interface by using socket programming for the client and server communication implemented in this project. On the remote a client is started and a receive task is created. On the gateway the server is created and another task is used for receiving.

Figure 11. Client/Server model



The remote device invokes the following command in the main loop:

```
/* Send ETH_MSG_3 */
eth_tx_buf[0] = ETH_MSG_3;
for (i = 1; i <= ETH_DATA_LEN; i++)
    eth_tx_buf[i] = (uint8_t)'a' + i;
eth_send_message(eth_tx_buf, ETH_DATA_LEN + ETH_HEADER_LEN);
LOG_ETH_MSG(0, (uint8_t)'T', ETH_MSG_3, eth_tx_buf, 8);
```

The data will be sent on the *gateway_socket* which is opened and connected since the beginning:

```
FreeRTOS_send(gateway_socket,&data[p], len, 0);
```

The eth_server will also receive the data (on the same port: 5555) and will invokes:
`send_to_gateway_from_task(&gw_msg);` to notify the gateway that a new message is available. According to the routing table, it is expected that this data will be routed to the network again.

```
/* ETH routing table */
RoutingTable_t eth_routing_table[] = {
…
    { ETH_MSG_3, ETH_TO_ETH, &ETH_DEVICE, ETH_MSG_4},
…
```

So the gateway will have to route the received packet to the client:

```
case
      ETH_TO_ETH:
```

```
LOG_MESSAGE("ETH TO ETH",
        gw_msg.id, dst, gw_msg.data_len, gw_msg.data);
```

```
send_to_eth(device, dst, gw_msg.data, gw_msg.data_len);
```

so:

```
n = FreeRTOS_send(eth_socket,
        &data[n], len - n, 0);
```

The packet will be received by the remote ethRecvTask (and dumped to the serial console).

8.4 Ethernet to CAN (or Lin)

According to the routing table, the remote device will send the messages from the Ethernet client to be routed either to CAN2 or to the LIN_MASTER.

```
RoutingTable_t eth_routing_table[] = {
    { ETH_MSG_1, ETH_TO_CAN, &CAN_DEVICE, CAN_MSG_3},
    { ETH_MSG_2, ETH_TO_LIN, &LIN_MASTER, LIN_MSG_3},
…
```

As soon as the server running on the gateway side receives the packet, it will notify the task to route the message as done in the previous scenarios. Routing the Lin message, this will be redirected to the LIN_MASTER instance that can be debugged by connecting an external protocol analyzer.

9 Conclusions

The main purpose of this application note and of the related demo application described is to show how to implement a complex gateway on a single core of the SPC584Cx/SPC58ECx 32-bit MCU. All the gateway and remote device software code can be easily ported on top of other SPC5x microcontrollers especially where the FreeRTOS operating system is available. The encapsulation of the messages among all the protocols mentioned in this application is not to be considered as a real user case, so further optimization, e.g. packaging or better buffer usage, could be adopted.

The eDMA could also be used for peripherals like LinFlex. For sure, the remote task could be replaced by an external hardware to stimulate the gateway (so the software should not spend any time to emulate a remote device). Satisfying these constraints, ad-hoc performance and latency measurements could be provided.

Appendix A UDP Logging

The rooting messages on the gateway log print messages through UDP frames.
The figure below shows how to configure the server and the UDP ports for this support.

Figure 12. UDP logging window

On the remote host, the user can have a console by using *netcat* under Linux or *udpterm_std* under Windows. UDP ports have to be configured as shown previously.

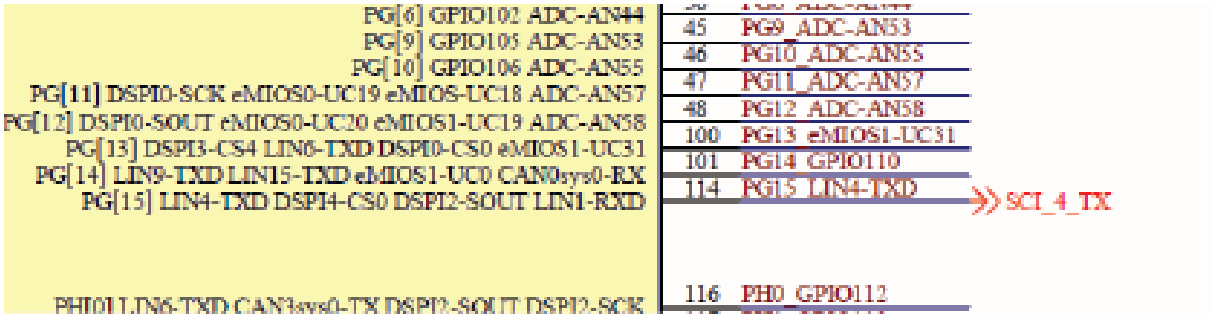
Appendix B UART console

In this project, one Linflex instance is configured as UART, here below you can find the SPC5Studio device initialization made by the remote device and the simple function to write a buffer on the UART:

```
void log_start(void)
{
    sd_lld_start(&SERIAL_DEVICE, NULL);
}
static void write_msg (char *msg, uint16_t len) {
    while (sd_lld_write(&SERIAL_DEVICE, (uint8_t *)msg, len) ==
    SERIAL_MSG_WAIT){}
}
```

The LinFlex4 used as UART is configured to have the TX on PG15 that is connected to the ST3232EBTR, so the console can be out by using USB mini port (configured as 38400,N,8,1).

Figure 13. Serial port TX line



Appendix C Source files

This is the list of the files that implement the gateway application. These files can be found in the directory source so they are not present in the SPC5Studio component.

The remote device emulator is implemented in the *remote.c* file, while the whole routing table and callback are found in the *routing.c*. The gateway task is defined in the *gateway.c* and all the peripheral callbacks are implemented in separated C and header files as shown below.

C files:

- *gateway.c*
- *remote.c*
- *routing.c*
- *can.c*
- *eth.c*
- *lin_master.c*
- *lin_slave.c*
- *log.c*

Header files

- *can.h*
- *candefs.h*
- *device.h*
- *eth.h*
- *ethdefs.h*
- *gateway.h*
- *led.h*
- *lin_master.h*
- *lin_slave.h*
- *lindefs.h*
- *log.h*
- *remote.h*
- *routing.h*

Appendix D Acronyms and abbreviations

Table 3. **Acronyms**

Abbreviation	Complete name
SIUL2	System integration unit lite 2
GW	Gateway
ISR	Interrupt Service Routine
MAC	Medium Access Control
eDMA	Enhanced Direct Memory Access
MII	Media independent interface

Appendix E Reference documents

- SPC58EHx/SPC58NHx 32-bit Power Architecture microcontroller Reference manual
- SPC58EHx/SPC58NHx Errata sheet, Datasheet and IO map excel file
- AN5416 SPC58EC CAN bus configuration

Revision history

Table 4. **Document revision history**

Date	Version	Changes
03-Sep-2020	1	Initial release.

Contents

List of tables

List of figures