Dual MCU Sync Solution User Guide

Table of Contents

1. Introduction

NXP S32K1xx serial MCU is widely used in automotive body control and many general-purpose automotive applications, while to target some applications with special requirements such as requiring more peripherals instance than the portfolio can offer (e.g. 6x CAN-FD, 6x LIN or 4 I2C) like mid-end BCM or DCU, an on-board dual/multi MCU sync solution is proposed as an alternate solution to extend the S32K1xx MCU peripherals/memory resource and CPU process capability.

The **eRPC** (**E**mbedded **R**emote **P**rocedure **C**all) is a Remote Procedure Call (RPC) system created by NXP(https://github.com/EmbeddedRPC/erpc/). An RPC is a mechanism used to invoke a software routine on a remote system using a sample local function call.

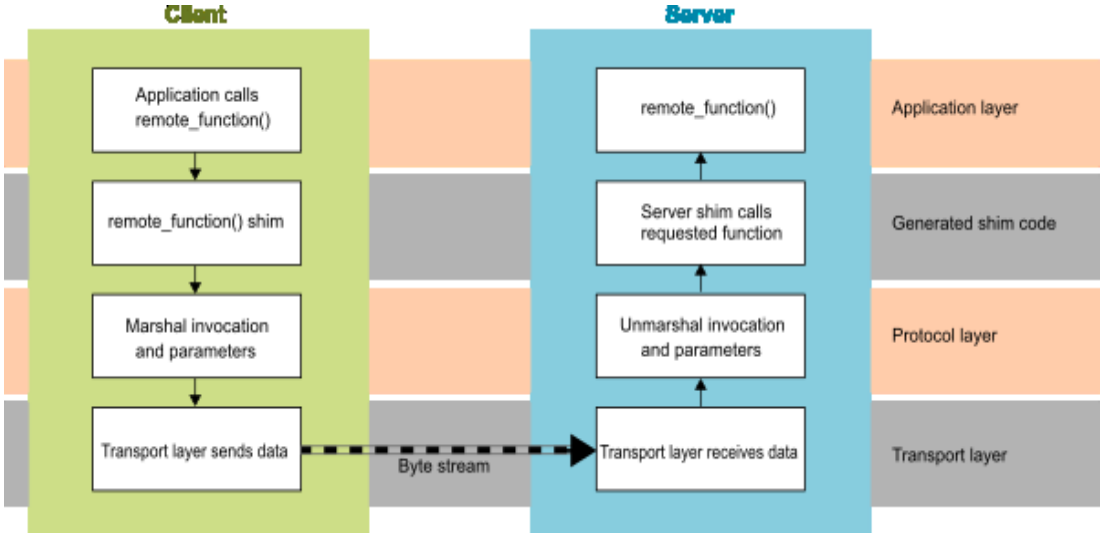


Figure 1. eRPC software architecture

In this project, we ported the eRPC protocol to S32K1xx platform, tested and figured out its performance. An out-of-box software package with detailed user guide (this document) is provided to simplify and accelerate users' assessment of eRPC on S32K1xx.

Two S32K144EVB boards are connected to demonstrate the usage of the eRPC protocol. One works as the client, another as the server. The client board starts an eRPC request and the server board responds to the request and executes the service.

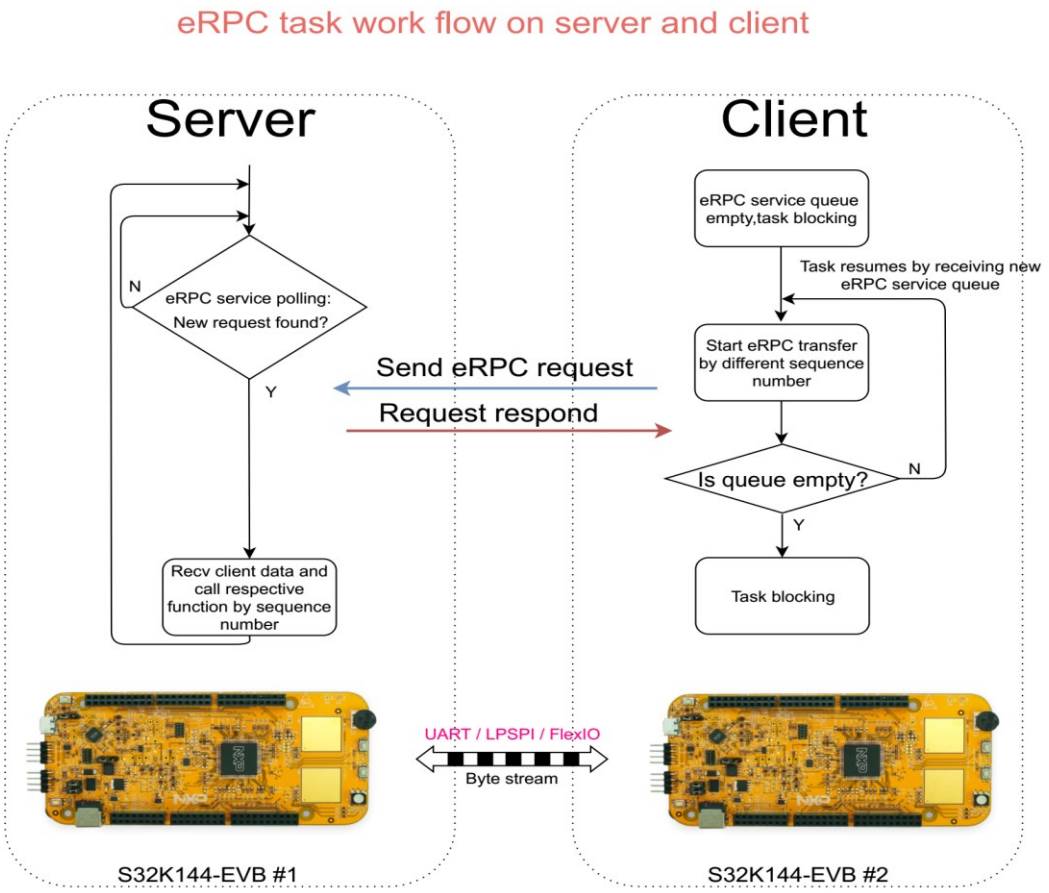


Figure 2. eRPC task workflow on S32K144-EVB

There are **three** types of MCU extensions are demonstrated in the project:

- ✓ **MCU IO extension**: Set LED;
- ✓ **MCU peripheral extension**: CAN and LIN message forwarding, LED luminance regulator;
- ✓ **CPU process capability extension**: Matrix multiply and addition math operation.

2. Requirements and Setup of S32K1xx eRPC Sample Projects

2.1 Hardware Requirements and Setup

To run the example, you need have the following items:

- ✓ **2x** S32K144EVB-Q100 boards
- ✓ **1x** power adapter 12V
- ✓ **1x** personal computer
- ✓ **1x** PCAN-USB adaptor (or any other CAN to USB adaptor for CAN message monitor/generation)

- ✓ **1x** Micro-USB cable
- ✓ several Dupont cables

Connections between 2 boards when using **UART**:

Server Board	Header	Client Board	Header
PTA2 (LPUART0.RX)	J1.1	PTA3 (LPUART0.TX)	J1.3
PTA3 (LPUART0.TX)	J1.3	PTA2 (LPUART0.RX)	J1.1
12V		12V	
GND		GND	

Table 1. UART signals connection of S32K144-EVB

Connections between 2 boards when using **SPI**:

Server Board	Header	Client Board	Header
PTA9 (LPSPI2.CS)	J6.19	PTA9 (LPSPI2.CS)	J6.19
PTE15 (LPSPI2.SCLK)	J5.4	PTE15 (LPSPI2.SCLK)	J5.4
PTE16(MISO)	J5.2	PTA8(MISO)	J6.17
PTA8(MOSI)	J6.17	PTE16(MOSI)	J5.2
PTA1(isSlaveReady)	J5.5	PTA1(isSlaveReady)	J5.5
12V		12V	
GND		GND	

Table 2. SPI signals connection of S32K144-EVB

Connections between the server board and PCAN-USB adaptor:

Server Board	PCAN-USB adaptor CH1
CAN_H	CAN_H
CAN_L	CAN_L
LIN	LIN
12V	VBat_LIN
GND	GND

Table 3. CAN and LIN signals connection between the server board and adaptor

Connections between the client board and PCAN-USB adaptor:

Client Board	PCAN-USB adaptor CH2
CAN_H	CAN_H
CAN_L	CAN_L
LIN	LIN
12V	VBat_LIN
GND	GND

Table 4. CAN and LIN signals connection between the client board and adaptor

Below is a photo snap of the real boards.

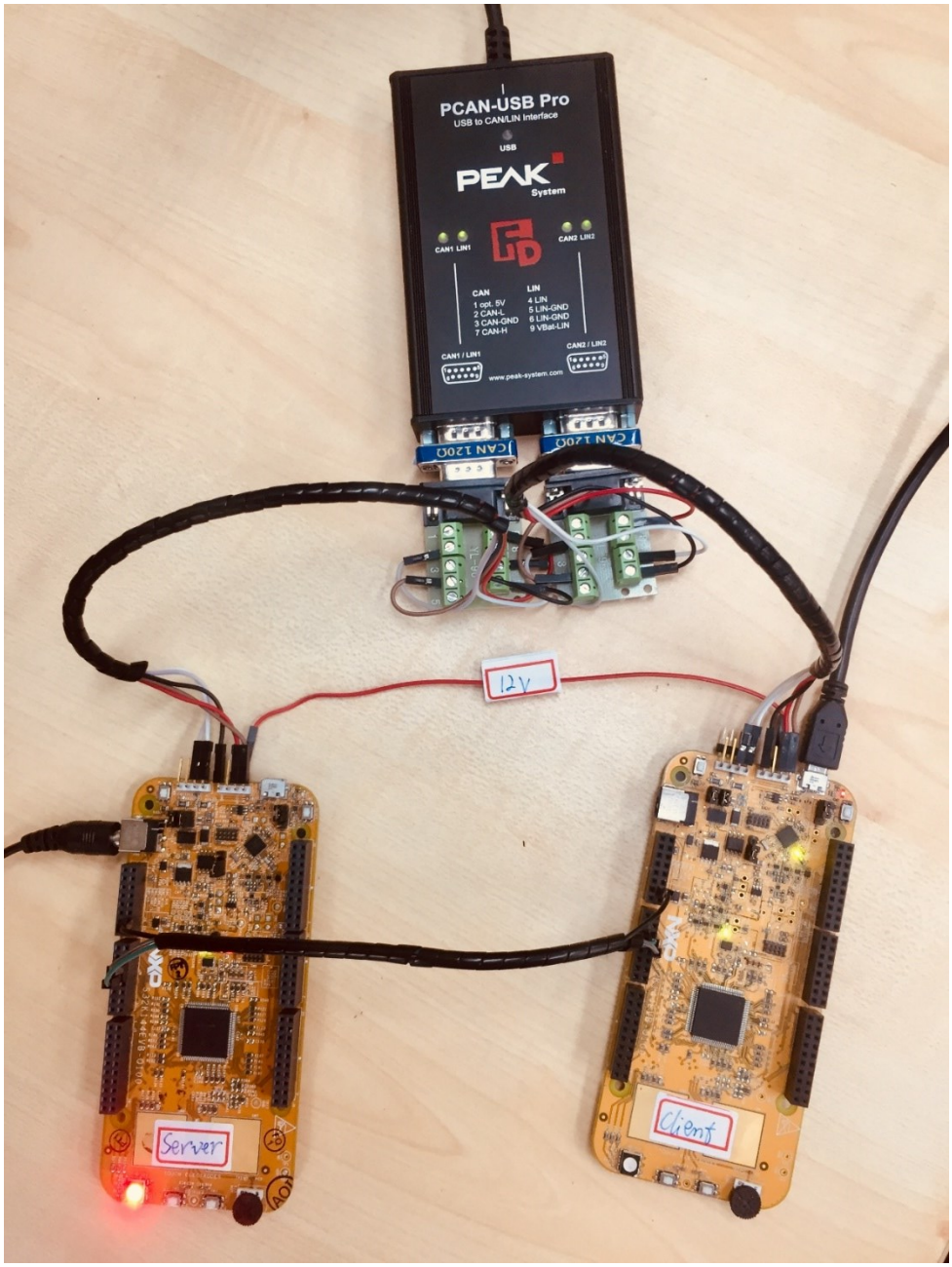


Figure 3. the photo snap of the two S32K144EVB boards eRPC system

2.2 Software Requirements and Setup

Software requirements:

- ✓ **IDE**: S32DS for ARM v2018.R1
- ✓ **SDK**: S32K1xx SDK RTM 3.0.

Steps to import the eRPC sample projects:

1. Open **S32DS for ARM 2018.R1 IDE**, click menu **File → Import → General → Existing Projects into Workspace → Select archive file → Browse** to the folder where your archive is, click **Finish**, import below the 3 projects at once.

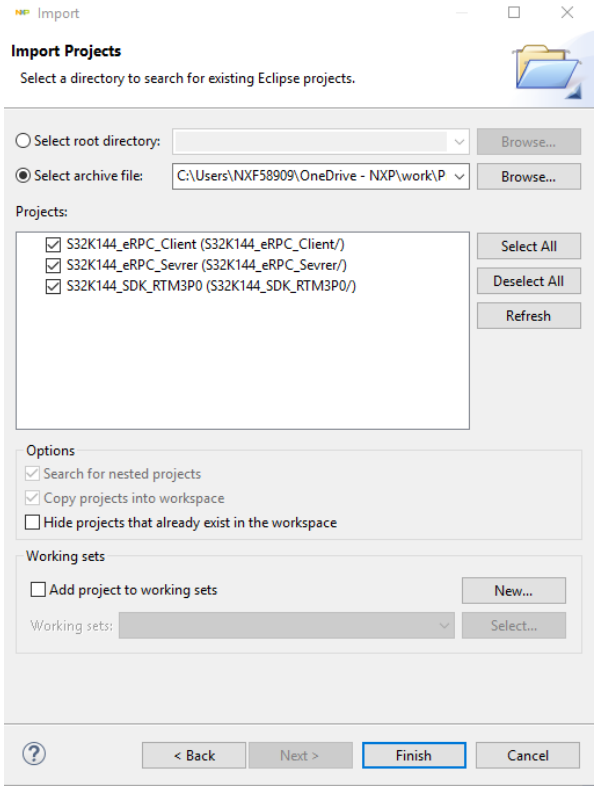


Figure 4. S32DS project importing

2. Open **Project→ Properties → Project References** to refer the C library project (**S32K144_SDK_RTM3P0**) to the two C++ application projects (**S32K144eRPC_Client** and **S32K144eRPC_Server**) respectively, every time when compiling the application project, the library project will be compiled first.

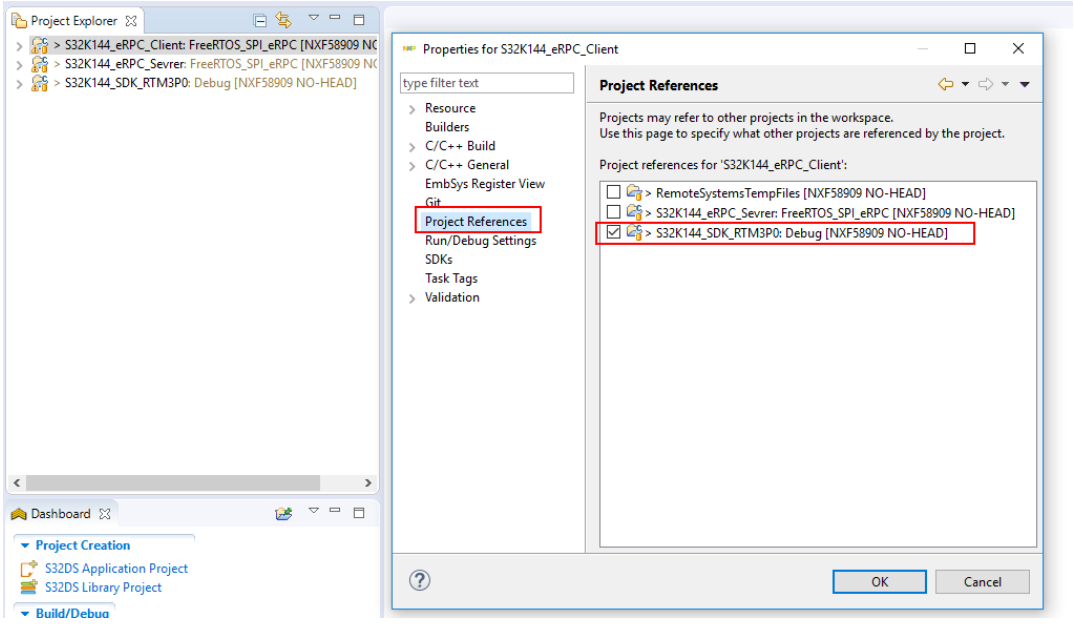


Figure 5. S32DS project References

3. Each project has 2 compile targets of UART and SPI version. Choose the version you want to build.

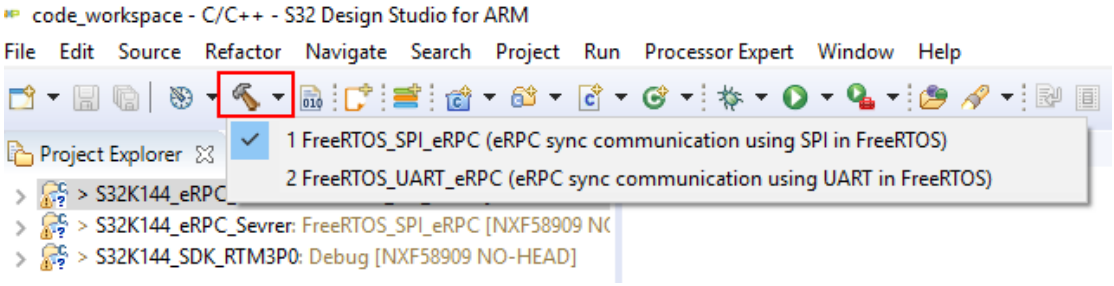


Figure 6. choosing different build target

3. eRPC Application Sample Project Tasks Implementation Details

In order to test and demonstrate the eRPC porting of S32K1xx, a sample project of eRPC for two S32K144-EVB sync over UART/SPI are created, and the following MCU extensions are implemented:

- ✓ **MCU IO extension**: Set LED;
- ✓ **MCU peripheral extension**: CAN and LIN message forwarding, LED luminance regulator;
- ✓ **CPU process capability extension**: Matrix multiply and addition math operation.

Details of the sample project on the eRPC client board and server board are explained in the following section.

Note : FreeRTOS 10.0.1 is used as the multi-task scheduler in the sample projects, all the eRPC services and remote function calls at the client side are implemented as FreeRTOS tasks to ensure the system's real-time performance, user can add their applications except eRPC related by creating a FreeRTOS task.

3.1 Tasks on the client board

The tasks on the eRPC client side/board are schemed as below:

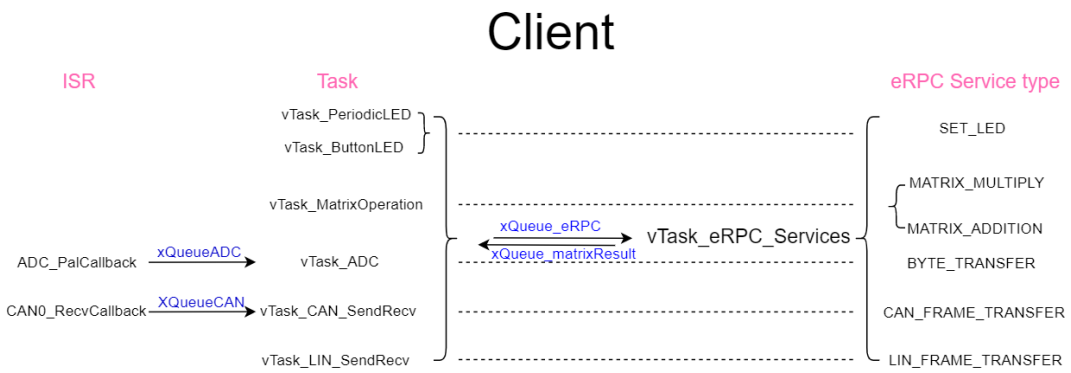


Figure 7. the eRPC client board task scheme

When the client hopes to call the eRPC service in different tasks, a queue(xQueue_eRPC) is sent and receives by the task(**vTask_eRPC_Request**), this task calls relevant eRPC service basing on different service type in the queue.

- ✓ **Task name**: **vTask_eRPC_Request**
- ✧ **Function**: Receive the eRPC queues from other tasks and request relevant eRPC service. It is set with the highest priority to ensure the eRPC service is called as fast as possible.

- ✓ **Task name: vTask_LocalLED**
- ✧ **Function**: The client board toggles its Green LED periodically to indicate the system is running.

- ✓ **Task name: vTask_ButtonLED**
- ✧ **Function**: Press button SW2 on the client board to toggle Blue LED on the server Board. For better observing the result, the vTask_LocalLED task in the server project can be closed.

- ✓ **Task name: vTask_PeriodicLED**
- ✧ **Function**: The blue LED on the server board toggles periodically by eRPC service sent from the client board. Press SW3 on the client board can suspend or resume this task. For better observing the result, we suggest closing the vTask_LocalLED task in the server project.

- ✓ **Task name: vTask_MatrixOperation**
- ✧ **Function**: The client board sends the parameters to the server board, gets the matrix multiply and addition result from the server and prints it out using LPUART2. Connect the Micro-USB cable to PC and open the Serial Debug Assist to see the result. UART communication setting: 115200, 8-bit/char, none parity, 1-bit STOP.

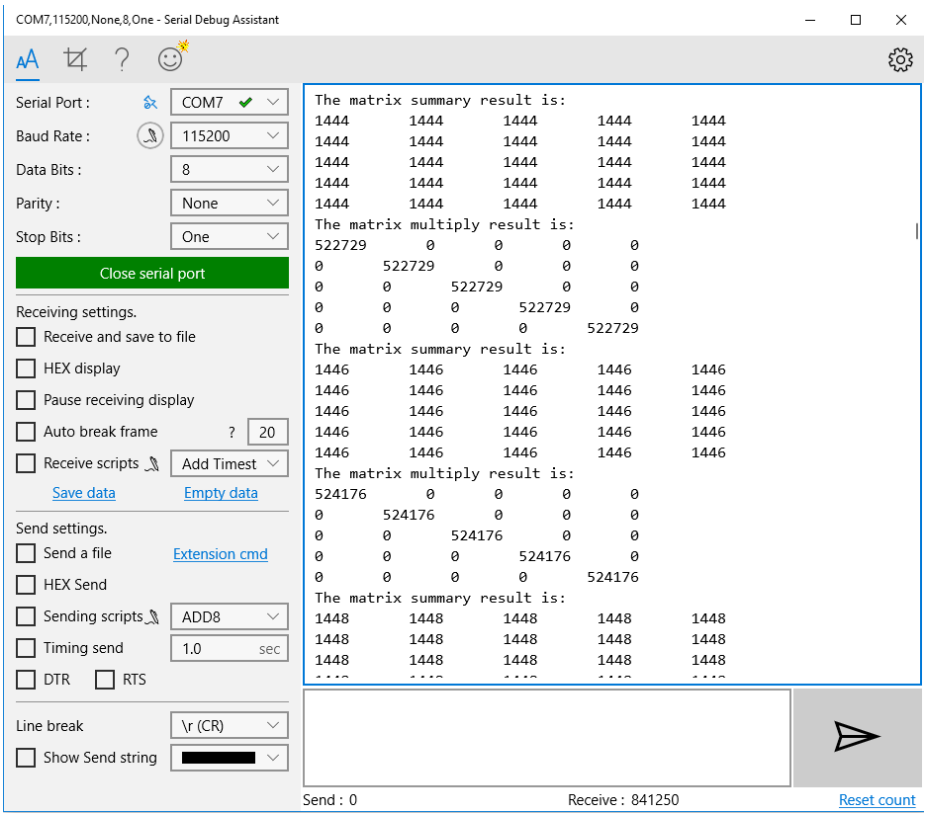


Figure 8. matrix operation printing result

✓ **Task name: vTask_ADC**

✧ **Function**: Rotate the potentiometer on the client board, the ADC convert result will be sent to the server via eRPC, the Red LED on the server board adjusts the luminance by PWM basing on the ADC value from the client board accordingly.

✓ **Task name: vTask_CAN_SendRecv**

✧ **Function**: Send any frame from the PCAN-View panel (Channel 2) to the client board. The client board receives the CAN frame and sends a CAN transfer eRPC service to the server board with the received CAN information. The server sends out the CAN frame.

✧ **Set up**: Open 2 PCAN-View software, connect to Channel 1 and 2 respectively. Set the baud rate to **500kBit/s**. Send one frame by double click or set as periodical transmit.

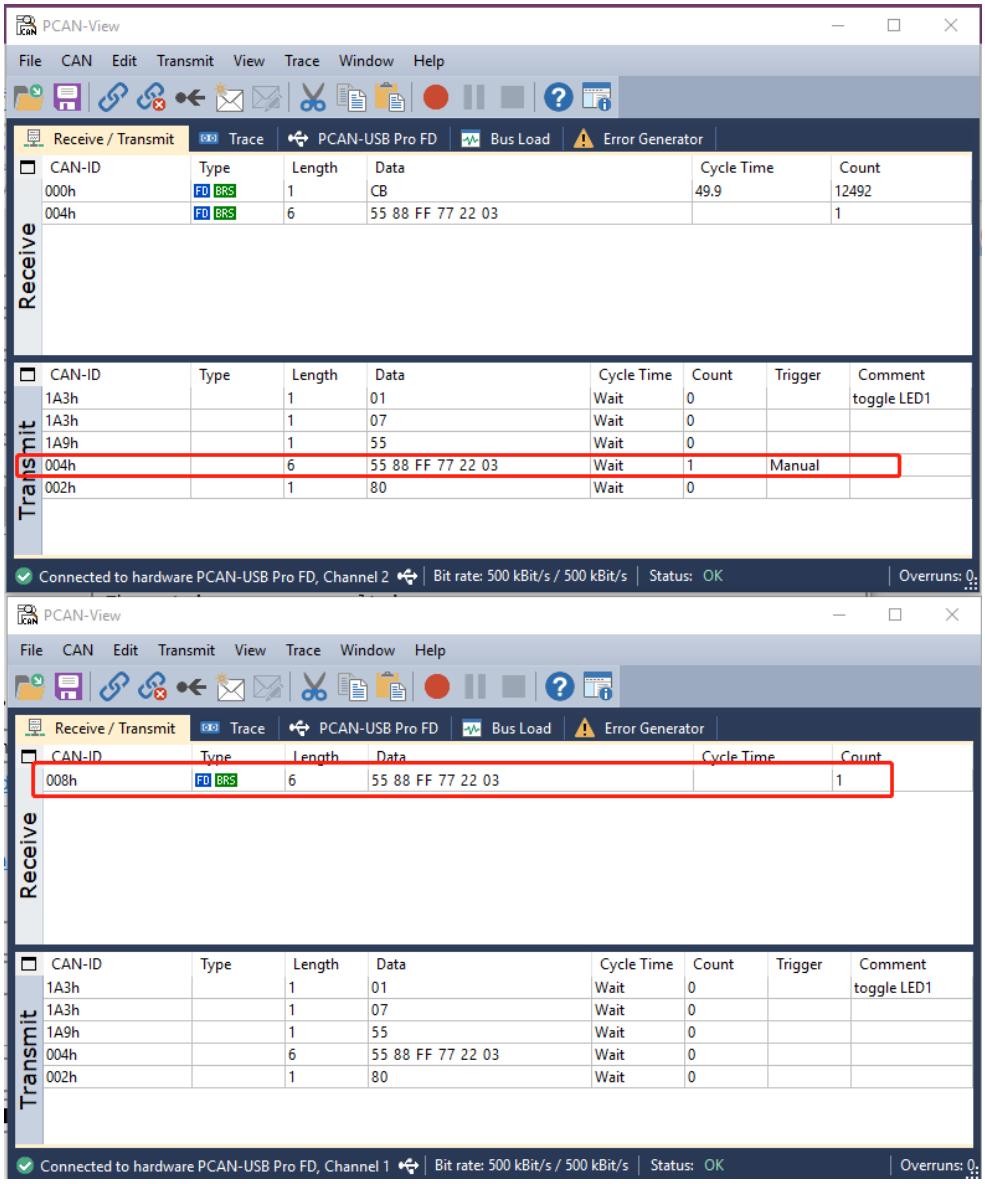


Figure 9. the eRPC CAN2.0 frame forwarding result

- ✓ **Task name: vTask_LIN_SendRecv**
- ✧ **Function**: Send a LIN frame with ID = **0x33**, 6 bytes playload data on the PLIN-View panel(Channel 2). The client board forwards the received LIN message frame to the server board via an eRPC request. The server board then send out a LIN frame of ID = **0x30**, length 1 byte, data varies with Motor1_temp(the first byte of the received LIN frame). If Motor1_temp >= 200, data = 0xFF. If 100 < Motor1_temp <200, data = 0xFE. Otherwise data = 0xFD.
- ✧ **Set up**: Open 2 PLIN-View software, connect to Channel 1 and 2 respectively. Set both panels to Slave mode and Bit rate 9600. On the panel of Channel 2 which connects to the client board, send the frame by left clicking, then press Space. You should be able to see the data changed on the panel of Channel 1 accordingly.

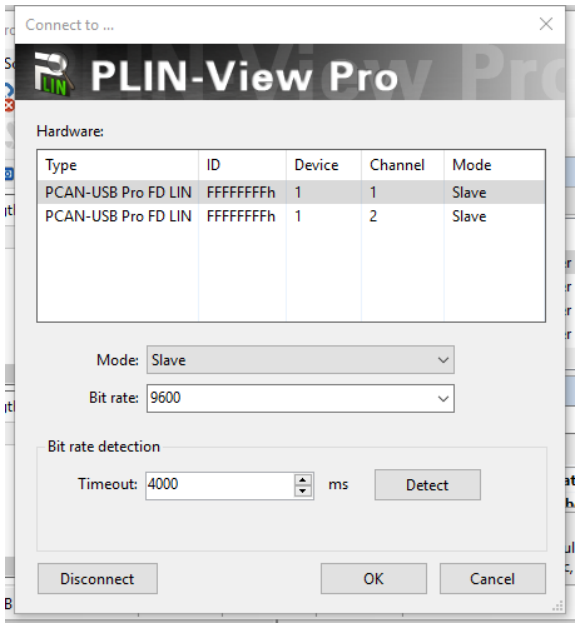


Figure 10. the PLIN-View setting

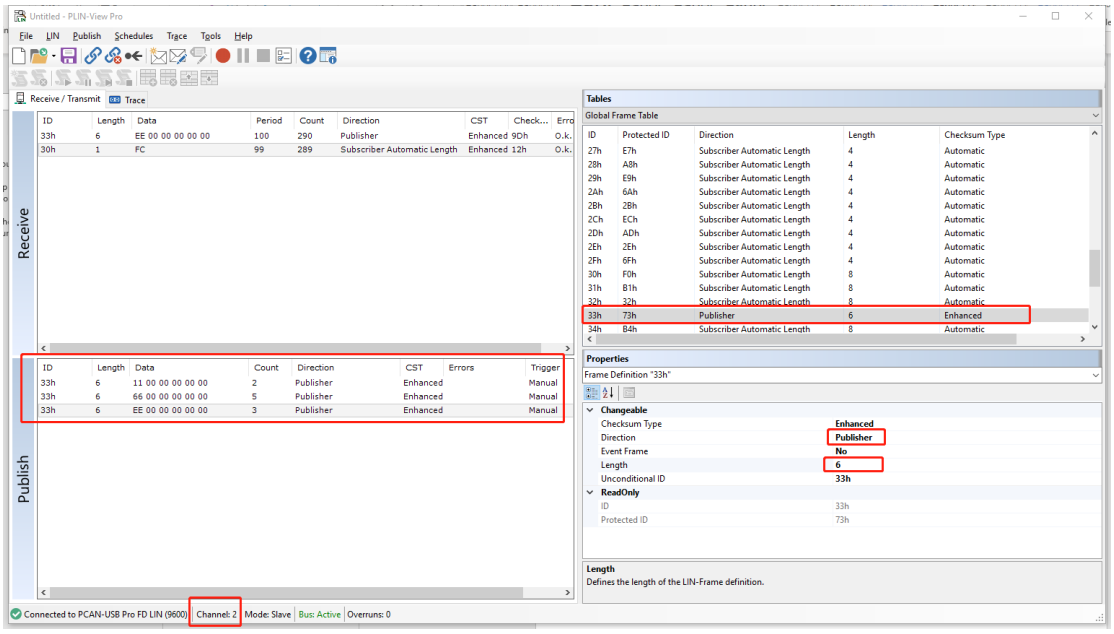


Figure 11. the PLIN-View setting on Channel 2

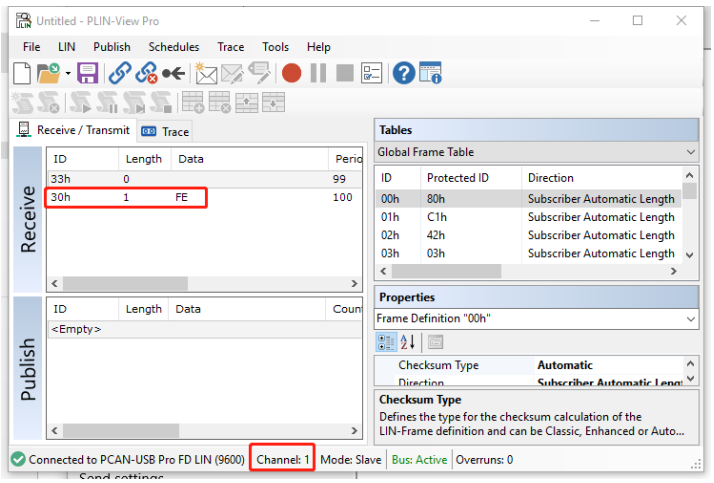


Figure 12. the PLIN-View setting on Channel1

3.2 Tasks on the server board

The tasks on the eRPC server side/board are schemed as below:

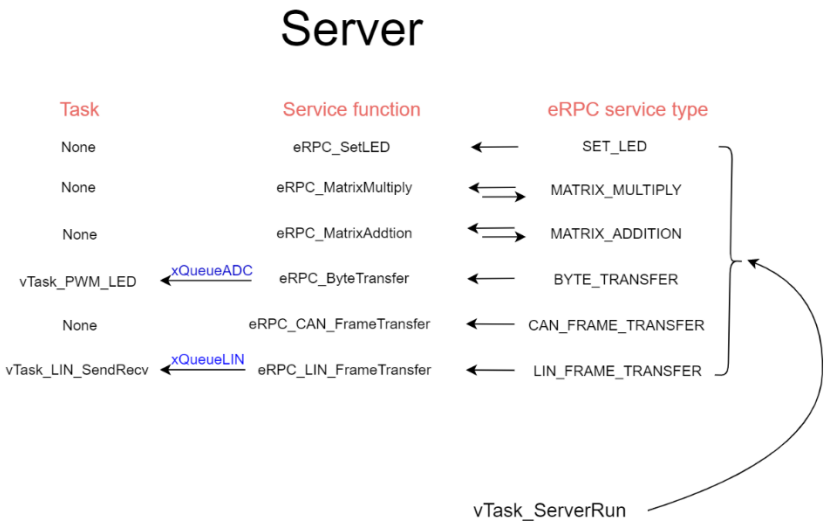


Figure 13. the eRPC server board task scheme

The eRPC server is always listening if there is any new request in task(**vTask_ServerService**). When there is service, it executes the service in the service function directly or sends a queue to relevant task to execute.

- ✓ **Task name: vTask_ServerService**
- ✧ **Function**: Receive the eRPC requests and call relevant service functions.

- ✓ **Task name: vTask_LocalLED**
- ✧ **Function**: The server toggles its Green LED periodically to indicate the system is running.

✓ **Task name: vTask_PWM_LED**

✧ **Function**: Adjust the luminance of the red led by PWM based on the ADC value sent from the client board accordingly.

✓ **Task name: vTask_CAN_SendRecv**

✧ **Function**: Forwarding the CAN frame received by the client board.

✓ **Task name: vTask_LIN_SendRecv**

✧ **Function**: Send a LIN frame basing on the LIN frame received by the client board requested by eRPC.

4. eRPC Protocol Performance

4.1 eRPC Protocol resolving

The eRPC protocol uses length-determined and CRC codec non-ACK communication mechanism. The server receives 4 bytes of data which contains the message length (LEN) and checksum (CRC) with codec before receiving the actual request. A determined length (LEN) of data will be transmitted afterward. The server sends the response to the client after service execution. The client receives the reply message using the same way. If the message type is oneway, no reply message from the server is needed.

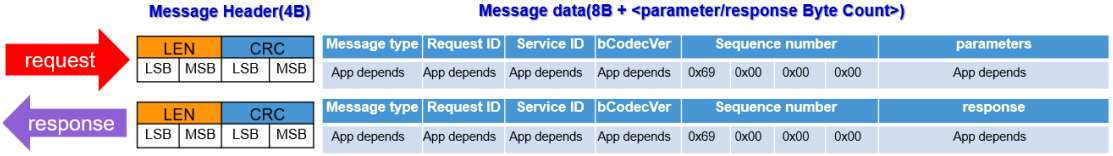


Figure 14. the eRPC message resolving

Definitions

✧ **Message type**: Indicate the type of the coming message.

```
typedef enum _message_type
{
    kInvocationMessage = 0,
    kOnewayMessage,
    kReplyMessage,
    kNotificationMessage
} message_type_t;
```

✧ **Request ID**: Generated by the IDL tool which is set to a fixed value 1.

✧ **Service ID**: Indicate different service type. Generated by the IDL tool according to the service defined in the IDL file and ranks automatically.

✧ **bCodecVer**: Fixed value 1.

✧ **Sequence number**: This number will automatically increase by one after each service is executed. The response message should have the same sequence number as the request.

✧ **Parameters**: The parameters transmitted from the client to the server. It's dependent to different services type.

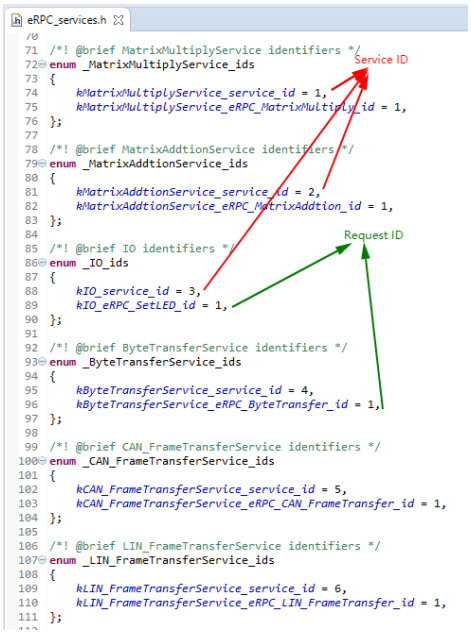


Figure 15. the eRPC Service ID and Request ID definition

Below is the example of an actual **eRPC_SetLED** service:

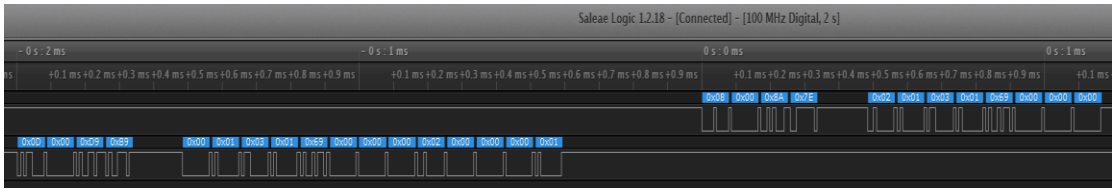


Figure 16. data stream of eRPC_SetLED service

Client send request: 0D 00 D9 B9 00 01 03 01 69 00 00 00 02 00 00 00 01
Server response: 08 00 8A 7E 02 01 03 01 69 00 00 00

Explanation

➢ The message from the client:

0x000D: The byte length of the coming message.
0XB9D9: The CRC value of the coming message.
0x00: The message type is **kInvocationMessage**.
0x01: The Request ID is 1.
0x03: The Service ID is **eRPC_SetLED**.
0x01: bCodecVer.
0x00000069: The Sequence number.
0x00000002: Parameter **whichled**.
0x01: Parameter **onOrOff**

➢ The message from the server:

0x0008: The byte length of the coming message.
0x7EA8: The CRC value of the coming message.
0x02: The message type is **kReplyMessage**.
0x01: The Request ID is 1.
0x03: The Service ID is **eRPC_SetLED**.
0x01: bCodecVer.
0x00000069: The Sequence number.

Note: All data type is LSB sent at first due to the eRPC using **little-endian** mode.

4.2 Latency of eRPC service remote call calculation

The whole time of a typical eRPC service request includes:

1. data transmission (dual way),
2. CRC calculation (dual way),
3. Codec (dual way),
4. Service API execution time on the eRPC server.

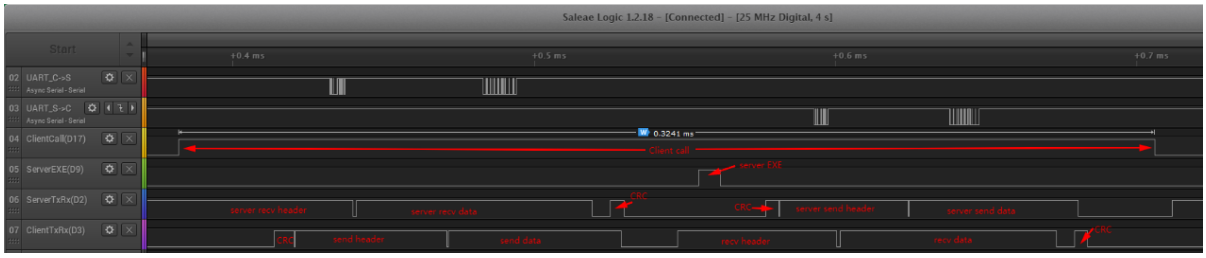


Figure 37. the timing of eRPC service request

Test IO implementation for the ClientCall (D17) to measure **the client calling time** at the eRPC client side.

*user_task_client.c

```
80      for(;;)
81      {
82          xStatus = xQueueReceive(xQueue_eRPC, &eRPC_DataRecv, OSIF_WAIT_FOREVER);
83      ①   PINS_DRV_SetPins(PTD,1<<17U);/*set PD17 high to indicate client start calling eRPC service*/
84          switch (eRPC_DataRecv.serviceType)
85          {
86          case SET_LED:
87              eRPC_SetLED(eRPC_DataRecv.whichLed, eRPC_DataRecv.onOrOff);
88              break;
89          case MATRIX_MULTIPLY:
90              eRPC_MatrixMultiply(eRPC_DataRecv.matrixA,eRPC_DataRecv.matrixB,resultMatrix);
91              xStatus = xQueueSend(xQueue_matrixResult,&resultMatrix,1000);
92              break;
93          case MATRIX_ADDITION:
94              eRPC_MatrixAddtion(eRPC_DataRecv.matrixA,eRPC_DataRecv.matrixB,resultMatrix);
95              xStatus = xQueueSend(xQueue_matrixResult,&resultMatrix,1000);
96              break;
97          case BYTE_TRANSFER:
98              eRPC_ByteTransfer(eRPC_DataRecv.dataToSend);
99              break;
100         case CAN_FRAME_TRANSFER:
101             eRPC_CAN_FrameTransfer(&eRPC_DataRecv.CAN_frame);
102             break;
103         case LIN_FRAME_TRANSFER:
104             eRPC_LIN_FrameTransfer(&eRPC_DataRecv.LIN_frame);
105             break;
106         default:
107             break;
108         }
109     ②   PINS_DRV_ClearPins(PTD,1<<17U);/*set PD17 low to indicate client finished calling eRPC service*/
110     }
111 }
```

Figure 48. test IO implementation for the ClientCall duration

Test IO implementation for ServerEXE (**PTD9**) to measure **the server request execution time(T_{exe})** on the server project.

```
/* implementation of RGB led set function */
void eRPC_SetLED(LEDName whichLed, bool onOrOff)
{
①   PINS_DRV_SetPins(PTD,1<<9U);/*set PTD9 high to indicate server start executing request from client*/
    if(kRed == whichLed)
        PINS_DRV_WritePin(PTD, 15, onOrOff);
    if(kBlue == whichLed)
        PINS_DRV_WritePin(PTD, 0, onOrOff);
    if(kGreen == whichLed)
        PINS_DRV_WritePin(PTD, 16, onOrOff);
②   PINS_DRV_ClearPins(PTD,1<<9U);/*set PTD9 low to indicate server finished executing request from client*/
}
```

Figure 59. test IO implementation for the server execution duration

4.3 Key factors of an eRPC service latency

According to the eRPC protocol, the key factors to affect an eRPC service latency include:

1. **Baud Rate**: Data transmission baud rate (MHZ);
2. **NBYTES**: Total transmitted data bytes;
3. **T_{exe}**: Server request execution time;
4. **Message type**: kOnewayMessage(oneway) or kInvocationMessage(non-oneway);

Other factors: CPU frequency, code optimization level, interrupt during process.

Transmission time is calculated as below:
One byte in UART is 10 bits which contains 8 data bits,1 start bit and 1 stop bit. So
$T_{transmission}$ = **NBYTES***10* (1/Baud Rate);

Note:
1. Message type will extend **the client calling time**, but the time from client calling service to server execution message won't be affected.
2. Length of the data bytes (**NBYTES**) will not only influence the transmission period, but also the CRC calculation and codec time.

4.4 Test Case Illustration

Test condition:

- ✓ **CPU clock**: 80MHZ;
- ✓ **Communication interface**: LPUART with DMA enabled and **8 Mbit/s baud rate**;
- ✓ **CRC type**: Hardware;
- ✓ **SDK library project optimization level**: -o1;
- ✓ **eRPC code (C++) optimization level**: -o1;
- ✓ **Application project optimization level**: -o0;
- ✓ **FreeRTOS tick**: 100us;

1. For SetLED service type, **29** bytes are transmitted in total.
   $T_{transmission}$ = NBYTES*10*(1/BaudRate) = 29*10*(1/8M) = 36 us.
   Measured client calling time is **318** us.

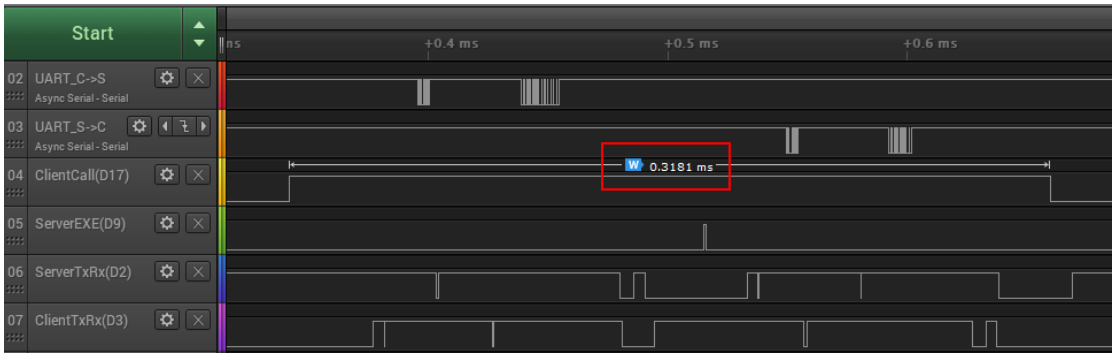


Figure 20. measured time for the ClientCall duration of SetLED service

2. For Matrix Multiply service type, **316** bytes are transmitted in total.
   $T_{transmission}$ = NBYTES*10*(1/BaudRate) = 316*10*(1/8M)= 395 us.
   Measured client calling time is **1442** us.

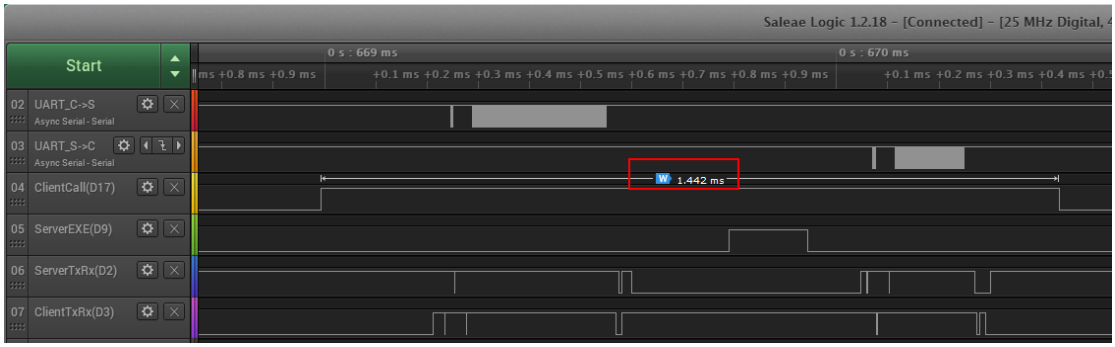


Figure 21. measured time for the ClientCall duration of the Matrix Multiply service

3. For CAN frame forwarding service type, **29** bytes are transmitted in total. Since it's oneway message type, client calling will return immediately without waiting for server execution and any response message.

   $T_{transmission}$ = NBYTES*10*(1/BaudRate) = 29*10*(1/8M) = 36 us.

   Measured client calling time is **207** us.

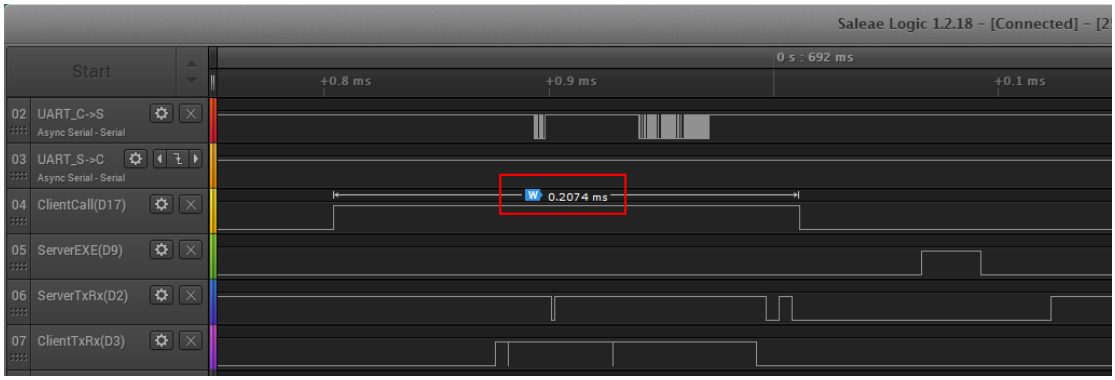


Figure 22. measured time for the ClientCall duration of the CAN frame forwarding service

4.5 Conclusion

Based on the above analysis and measurements, to improve the eRPC efficiency, the following tips can be used:

Key factors to improve	Tips	Notes
Baud Rate	Configure to use higher serial communication baud rate to reduce the message transmission time	Both SPI and UART are tested fine below baud rate 10Mbit/s. Data may be lost when speed is higher. And the data transmission time won't affect the whole eRPC too much after certain speed, so we suggest keeping the baud rate under 10Mbit/s.

NBYTES	Use smaller data type to reduce data bytes length needed to send/receive to reduce the message transmission time	For example, to control a LED on/off state, a byte(8-bit) parameter should be used instead of a word(32-bit) type.
T_{exe}	Improve CPU clock and set higher compilation level as well as use hardware CRC can reduce the execution time of the eRPC protocol process and remote service route	S32K14x support 112MHz HSRUN mode and hardware FPU for float data process acceleration.
Message type	Optimize the application to use oneway message type for time-critical remote service request.	Comparing to **kOnewayMessage**(oneway) type message, **kInvocationMessage**(non-oneway) type message needs to wait for server's response after remote service execution is finished and result is obtained.

Table 5. tips for improving the eRPC efficiency

5. eRPC Use Case Illustration--6x CAN gateway based on two S32K148 T-BOX boards

This sample project is built on two **S32K148 T-BOX** boards (MCU P/N: FS32K148UJT0VLQT) to demonstrate a typical eRPC use case—6x CAN gateway.

The following software environments are required to run the demo:

- ✓ **SDK**: S32K1xx RTM3.0;
- ✓ **OS**: FreeRTOS v10.0.1(included in the SDK package);
- ✓ **IDE**: S32DS for ARM v2018.R1;

2x S32K148-T-BOX board (as below figure) are needed to run the demo:

Figure 23. S32K148-T-BOX

The S32K148-T-BOX board routes S32K148's 3 CAN bus via a 23-pin ECU connector for the gateway CAN connectivity:

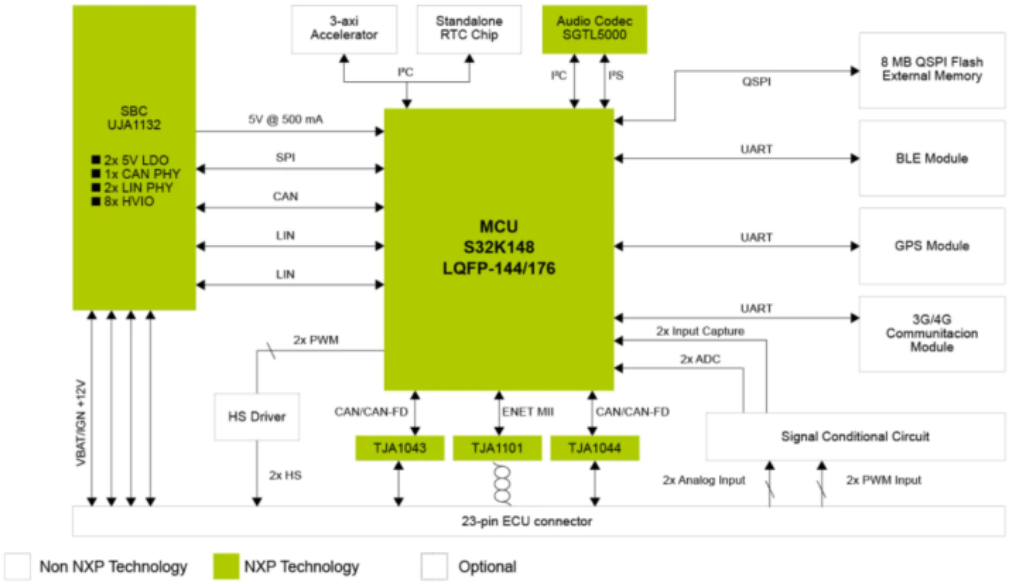


Figure 24. S32K148-T-BOX diagram

5.1 Function description

All the used 6x FlexCAN modules are configured with **500 Kbit/s** baud rate to transfer **CAN 2.0** message frame. A UART (based on LPUART with hardware flow control enabled, 7 Mbit/s baud rate) communication is used in transport layer for eRPC sync between the two S32K148 T-BOX boards.

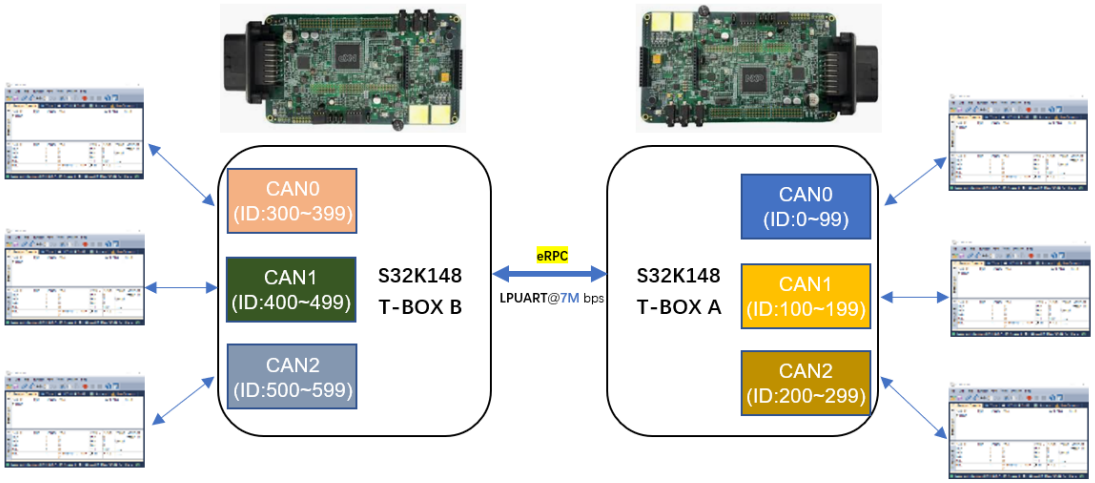


Figure 25. the eRPC system diagram basing on two S32K148-T-BOX

CAN message frame received from any of the 6 CAN bus/ports is transmitted/forwarded out by different CAN bus/port based on its 11-bit standard message frame ID as below table. For example, one CAN2.0 message frame of ID 120 is received from CAN0 on board B, an eRPC service will be called to send relevant CAN message to board A, then send out the CAN message by CAN1 on board A.

S32K148 T-BOX board	CAN #	Rx CAN Message ID	Tx/Forward CAN Message ID
A	**CAN0**	any	0~99
	CAN1	any	100~199
	CAN2	any	200~299
B	**CAN0**	any	300-399
	CAN1	any	400-499
	CAN2	any	500-599

Table 6. CAN ID layout for different CAN ports

5.2 Key codes

- The box B receives CAN frame in the callback then judge the ID. If the ID range is on the board itself, call relevant **CAN_Send ()** function, if the ID range is on the other board, send the CAN frame to the queue and call the eRPC service in the task which is set to the highest priority.
- The eRPC **vTask_Server** task on the box A which is set to the highest priority receives the eRPC request and call **CAN_Send ()** in the eRPC service function.
- For each FlexCAN module, 5x MB are configured for CAN message transmit and 3x MB are configured for CAN message receive.

user_task_box_B.c

```
void CAN0_RecvCallback(uint8_t instance,can_event_t eventType,uint32_t buffIdx,void *state)
{
    BaseType_t xResult,xHigherPriorityTaskWoken = pdFALSE;
    uint8_t index;
    index = CAN0_RX_BufferIndex%CAN_RX_Buffer_MAX;
    if(CAN_EVENT_RX_COMPLETE == eventType)
    {
        if(CAN0_RX_Message_Buffer[index].id >= MIN_ID_BRD_A_CAN0 && CAN0_RX_Message_Buffer[index].id < MAX_ID_BRD_A_CAN2)
        {/*if CAN ID range is belong to BOX A, send the CAN frame to box A by eRPC*/
            /*post the RX completed message/buffer through the queue at first*/
            xResult = xQueueSendFromISR(CAN_RX_Queue,&CAN0_RX_Message_Buffer[index],&xHigherPriorityTaskWoken);
        }
        else if(CAN0_RX_Message_Buffer[index].id >= MIN_ID_BRD_B_CAN0 && CAN0_RX_Message_Buffer[index].id < MAX_ID_BRD_B_CAN0)
        {/*if CAN ID range is belong to local board(BOX B), send the CAN frame by local CAN0*/
            CAN_Send(&can_pal0_instance,CAN0_TxMB++%CAN_TX_MB_USED, &CAN0_RX_Message_Buffer[index]);
        }
        else if(CAN0_RX_Message_Buffer[index].id >= MIN_ID_BRD_B_CAN1 && CAN0_RX_Message_Buffer[index].id < MAX_ID_BRD_B_CAN1)
        {/*if CAN ID range is belong to local board(BOX B), send the CAN frame by local CAN1*/
            CAN_Send(&can_pal1_instance,CAN1_TxMB++%CAN_TX_MB_USED, &CAN0_RX_Message_Buffer[index]);
        }
        else /* CAN ID which is more than MIN_ID_BRD_B_CAN2*/
        {/*if CAN ID range is belong to local board(BOX B), send the CAN frame by local CAN2*/
            CAN_Send(&can_pal2_instance,CAN2_TxMB++%CAN_TX_MB_USED, &CAN0_RX_Message_Buffer[index]);
        }
        /*use next circle buffer for the new CAN message receive*/
        CAN_Receive(&can_pal0_instance,buffIdx,&CAN0_RX_Message_Buffer[++CAN0_RX_BufferIndex%CAN_RX_Buffer_MAX]);

        if( xResult != pdFAIL )
        {
            /* If xHigherPriorityTaskWoken is now set to pdTRUE then a context
            switch should be requested.  The macro used is port specific and will
            be either portYIELD_FROM_ISR() or portEND_SWITCHING_ISR() - refer to
            the documentation page for the port being used. */
            portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
        }
    }
}
```

Figure 26. implementation code on board B CAN0 receive callback

```
/* implementation of CAN frame transfer from B->A function */
void eRPC_CAN_FrameTransfer(const CAN_message * CAN_frameRecvFromClient)
{
    uint8_t i;
    can_message_t sentMsg;
    sentMsg.length = CAN_frameRecvFromClient->length;
    sentMsg.id = CAN_frameRecvFromClient->id;
    sentMsg.cs = CAN_frameRecvFromClient->cs;
    for(i = 0;i < maxCanData; i++)
        sentMsg.data[i] = CAN_frameRecvFromClient->data[i];

    INT_SYS_DisableIRQGlobal();

    if(sentMsg.id >= MIN_ID_BRD_A_CAN0 && sentMsg.id < MAX_ID_BRD_A_CAN0)
        CAN_Send(&can_pal0_instance,CAN0_TxMB++%CAN_TX_MB_USED, &sentMsg);
    else if(sentMsg.id >= MIN_ID_BRD_A_CAN1 && sentMsg.id < MAX_ID_BRD_A_CAN1)
        CAN_Send(&can_pal1_instance,CAN1_TxMB++%CAN_TX_MB_USED, &sentMsg);
    else
        CAN_Send(&can_pal2_instance,CAN2_TxMB++%CAN_TX_MB_USED, &sentMsg);

    INT_SYS_EnableIRQGlobal();
}
```

Figure 27. implementation code of eRPC service on board A

It's similar from board A to B except the eRPC function name is slightly different.

5.3 Performance

Theoretically, with **500 Kbit/s** speed, a CAN2.0 **standard** (11-bit ID) message frame with 8 bytes data will take (44 + 8 * 8) * 1/**500KHz** = **216us**:

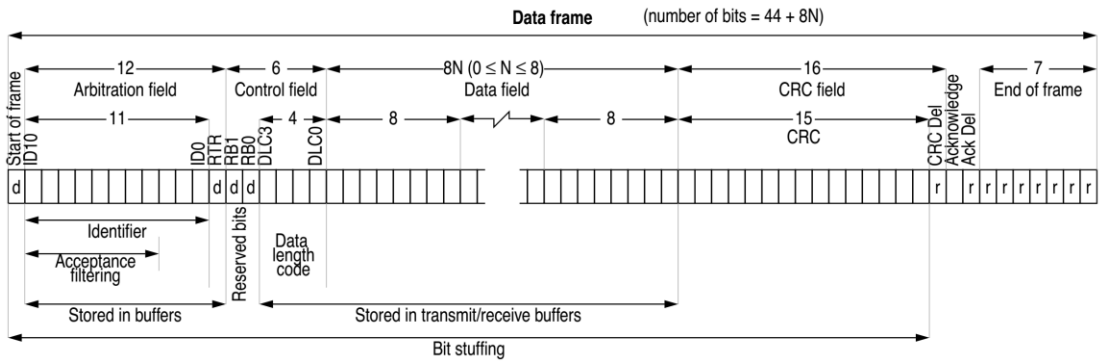


Figure 28. CAN2.0 frame bits definition

The eRPC can support maximum **4x** CAN2.0 standard message frame with 8 bytes data to be forwarded by one CAN port in 1ms.

With the arbitration feature is implemented in the eRPC porting for S32K148, it can also support eRPC request bidirectionally, that is to say the eRPC client and server can request CAN message forward via eRPC at the same time. For instance, 2(or 1) CAN messages received by board A can be forwarded to the board B, meanwhile another 2(or 3) CAN messages received by board B can be forwarded to the board A.

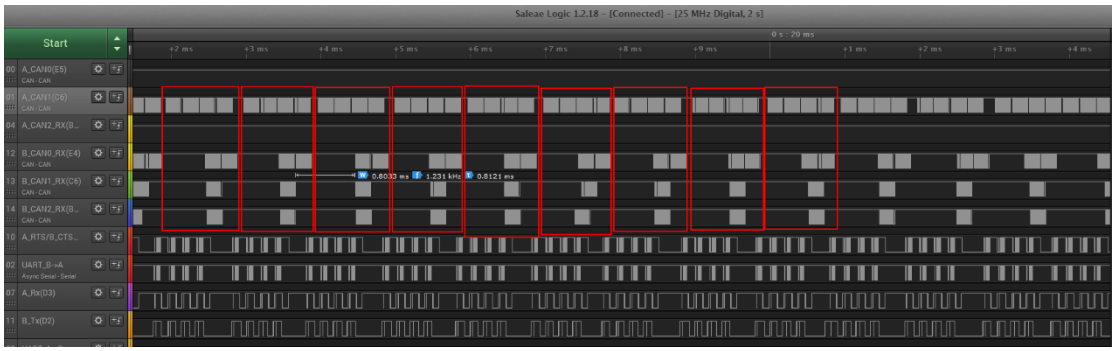


Figure 29. four CAN frames forwarding by eRPC in 1ms

The latency of forwarding is less than **500**us.

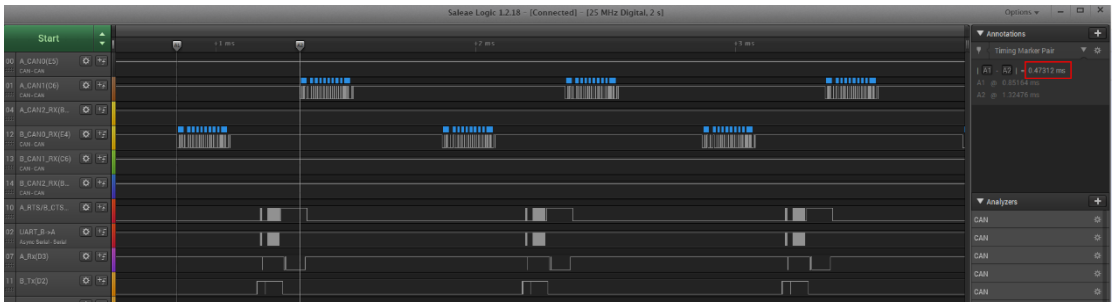


Figure 30. the latency of CAN forwarding

6. Summary

eRPC is an embedded system orientated lightweight, low-cost but high real-time performance dual MCU sync protocol written in C++, it can facilitate the remote application call implementation via a layered architecture and erpcgen.exe utility tool to generate the shim codes according to user's inputs through an easy-use **IDL** file.

As eRPC is a service request based server-client solution, at the server side, CPU needs to monitor the service request from the slave side constantly, so it requires an RTOS to schedule the multi-application tasks except the eRPC sync task for an actual application, that is to say, bare-metal eRPC implementation is useless. For S32K1xx family MCU, FreeRTOS integrated in S32K1xx SDK or AutoSAR OS offered in S32K1xx AutoSAR SW package can be used as the task scheduler with the eRPC.

Using the eRPC for S32K1xx porting package and demo projects, user can build their own application prototype with very few codes/configuration modifications and start to evaluate the eRPC performance.

Appendix A. Steps to Create an eRPC Service

This chapter introduces how to create the service shim code implementing on the client and server project.

1. Create the service in .erpc file

Open the .erpc file, define the service and the argument type needed. In this case, we want a set LED service, so define a LEDName enum data type. More IDL grammar explaination please refer to https://github.com/EmbeddedRPC/erpc/wiki/IDL-Reference.

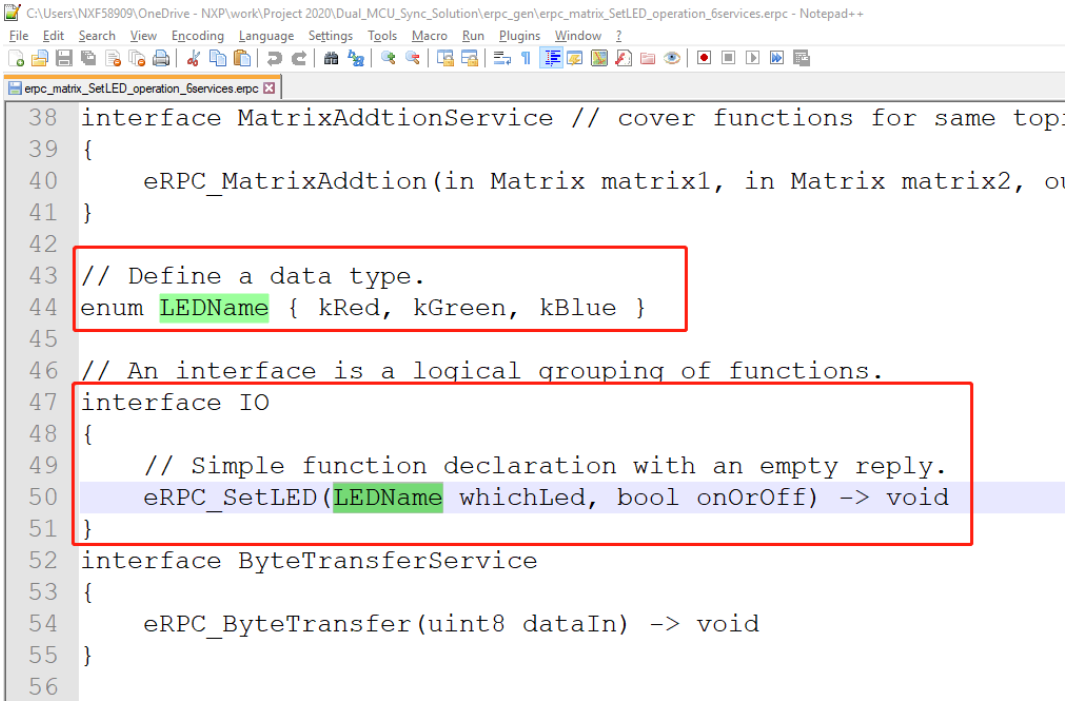


Figure 31. definition of eRPC_SetLED service using IDL language

2. Run erpcgen.exe to generation eRPC shim layer codes

Open the Windows command terminal, enter the folder where the erpcgen.exe and .erpc file located, run below command. Four source files are generated accordingly.

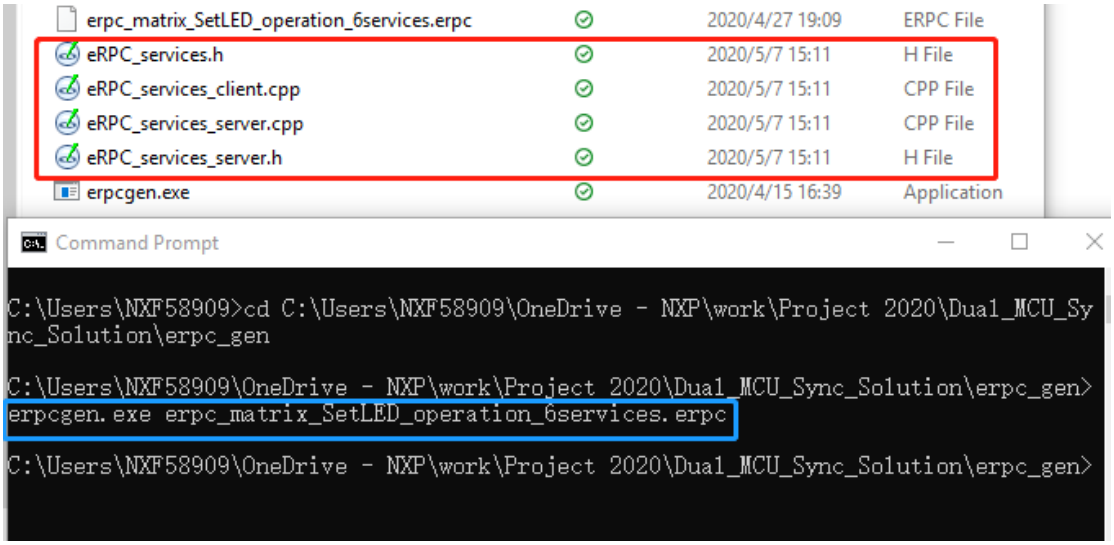


Figure 32. eRPC services source file generation from the IDL file

3. Add source file to project

Add file <**eRPC_services_client.h**>, <**eRPC_services_server.cpp**> and <**eRPC_services.h**> to the server project, add <**eRPC_services.h**> and <**eRPC_services_client.cpp**> to the client project respectively.

4. Implement the service function in the server project

Per application requirement, user can implement the service function in the server project as below for example:

```
69  /* implementation of RGB led set function */
70  void eRPC_SetLED(LEDName whichLed, bool onOrOff)
71  {
72      if(kRed == whichLed)
73          PINS_DRV_WritePin(PTD, 15, onOrOff);
74      if(kBlue == whichLed)
75          PINS_DRV_WritePin(PTD, 0, onOrOff);
76      if(kGreen == whichLed)
77          PINS_DRV_WritePin(PTD, 16, onOrOff);
78  }
```

Figure 33. the function implementation of eRPC_SetLED service

5. Adding service to server.

Call **erpc_add_service_to_server**(create_xxxx_service()) in the server project to add relevant service to server. Xxxx is the word after key word <interface> in .erpc file. For the set LED service, **erpc_add_service_to_server**(create_IO_service()) will be called.

6. Call the service in the client project.

Call eRPC_SetLED in the client project, the corresponding function on the server board will be called after receiving the eRPC message. For the client, just like calling a subroutine.

Appendix B. FAQ and Useful Tips

1. How to debug 2 boards simultaneously?

When debugging the second board after first board has been launched, change the **GDBMI and Server Port number** to another such as 6225 and 7225 of the relevant debug targets.

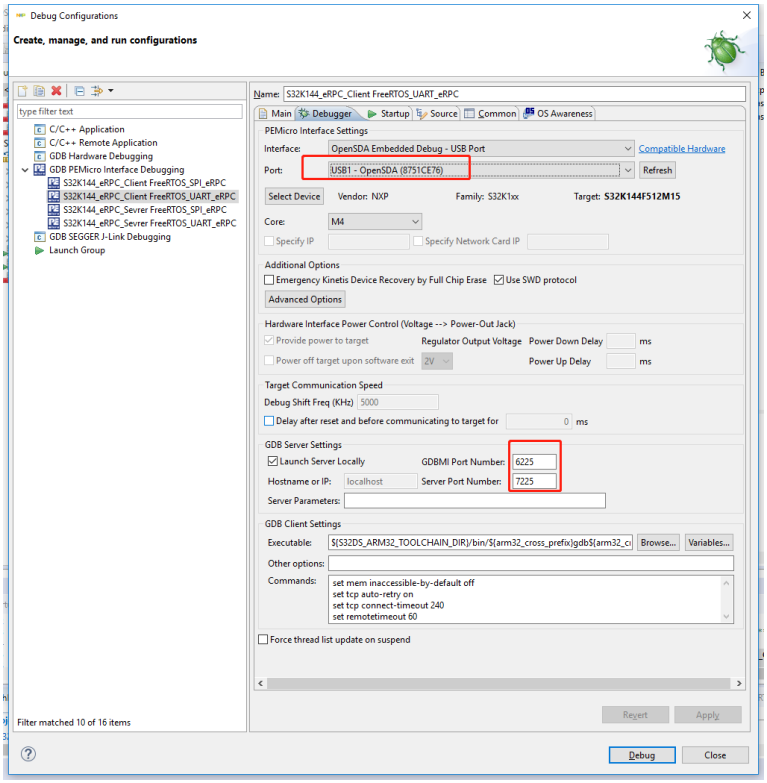


Figure 34. the GDB server setting

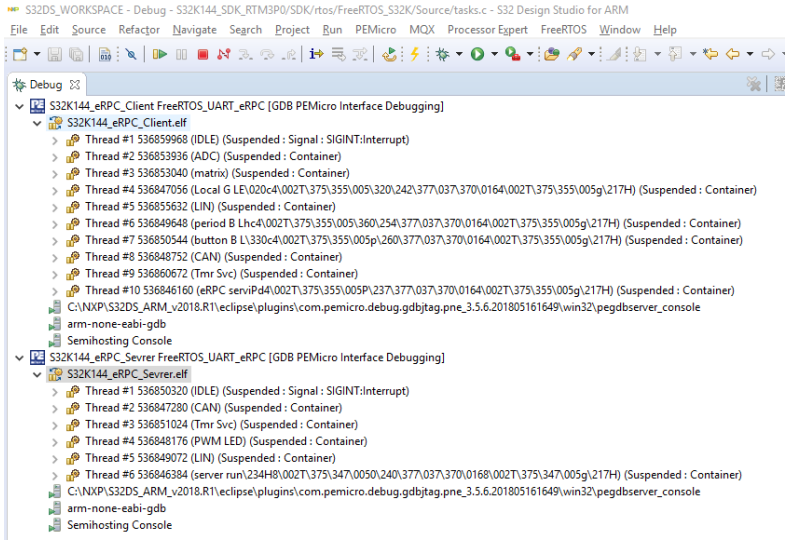


Figure 35. debugging 2 boards simultaneously in S32DS

2. How to create an eRPC based application project using S32DS from scratch?

The eRPC protocol is written in C++, so it requires a C++ application project (both C and C++ compilers will be involved, and C++ linker is used as the project linker to support C++ advanced features) while S32K1xx SDK supports only C compiler by default. As a result, two C++ application projects are needed for the eRPC applications, another C static library project with S32K1xx SDK RTM3.0 processor expert GUI enabled is used for peripherals initialization and generate a static library to be called in the two eRPC C++ application projects. With S32DS for ARM v2018.R1 IDE, user can create an eRPC based application project from scratch as the following steps:

1. **Create the C library project**.

(1). Click menu **File → New → S32DS Application Project**. Choose the S32K1xx part number you are using and name the project. In this case we use S32K148. Click **Next**.

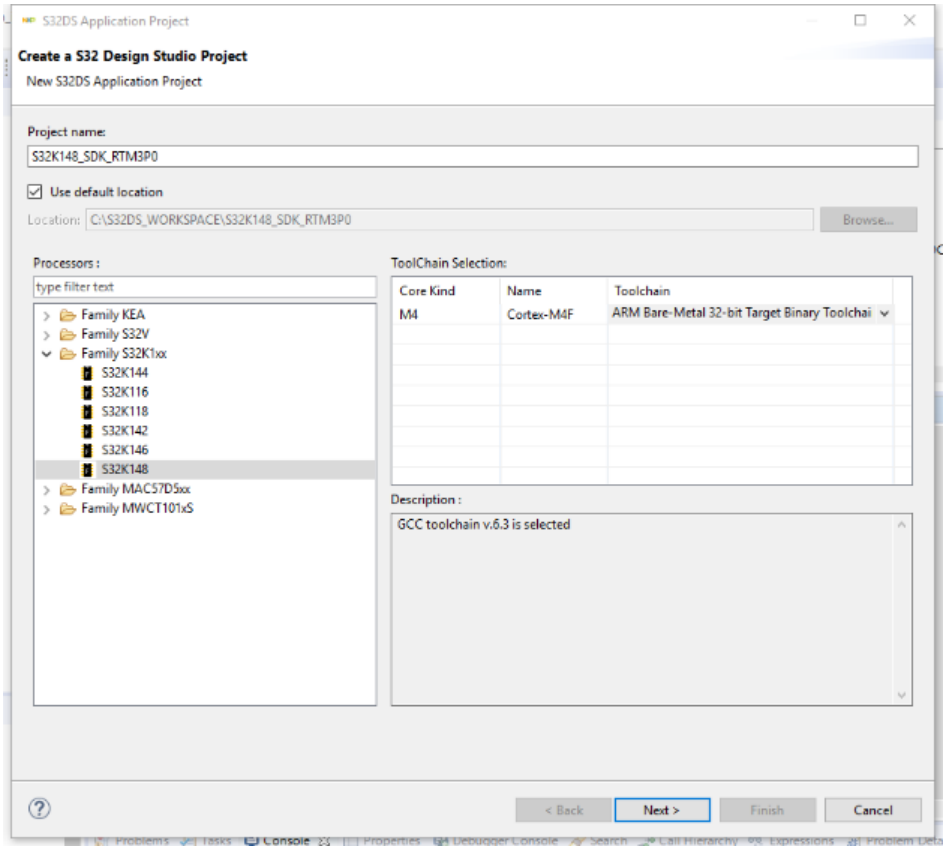


Figure 36. creating the C library project S32K148_SDK_RTM3P0

Tips: Creating an application project instead of a static library project should be used to enable S32K1xx SDK in Step (2), NO S32K1xx SDK options for a static library project.

(2). Select **C** as the programing langue, and **S32K148_SDK** RTM 3.0.0 for **SDKs**. Others stay as default.

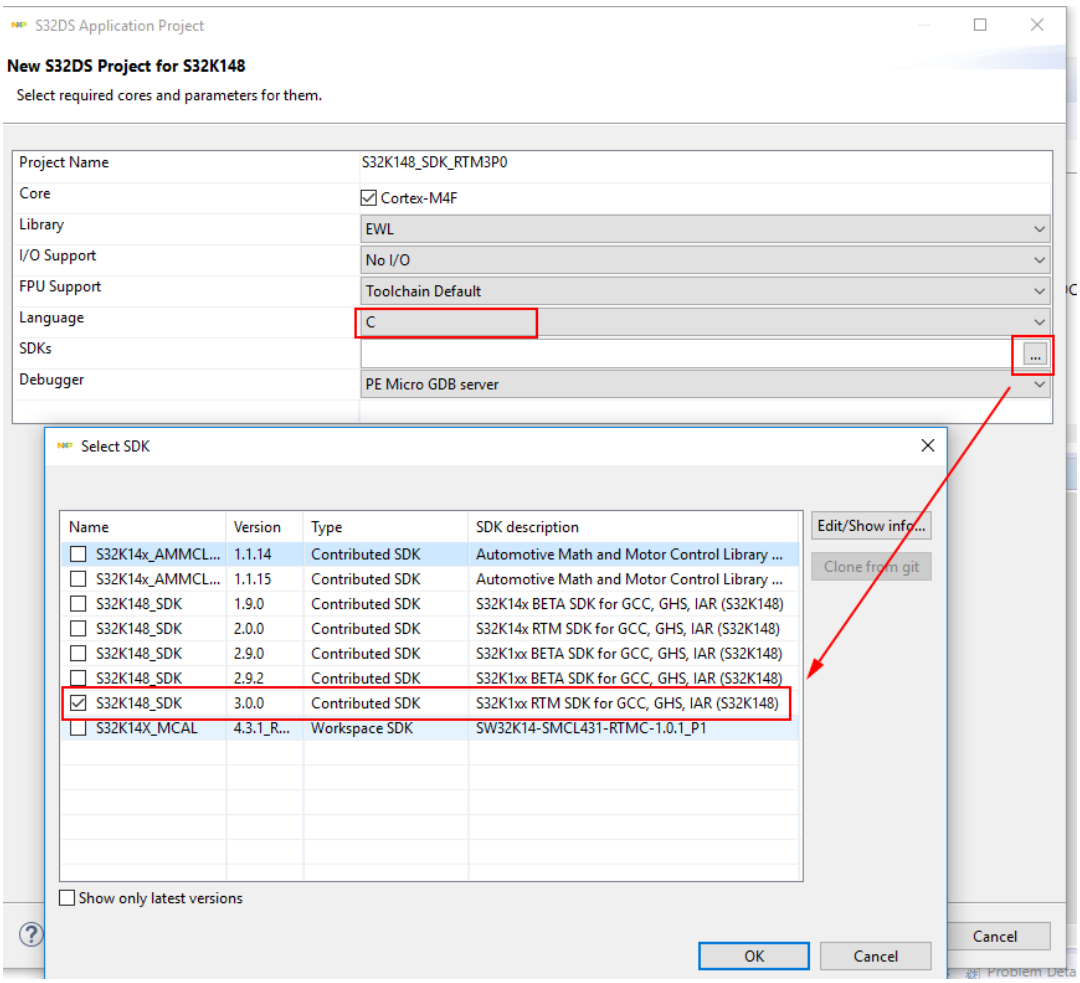


Figure 37. choosing project language and SDK in S32DS

(3) Click **Project → Property → C/C++ Build → Settings → Build Artifact → Artifact Type**, choose **ARM32 Library**. In this step, the project build target is changed from an executable binary to a library.

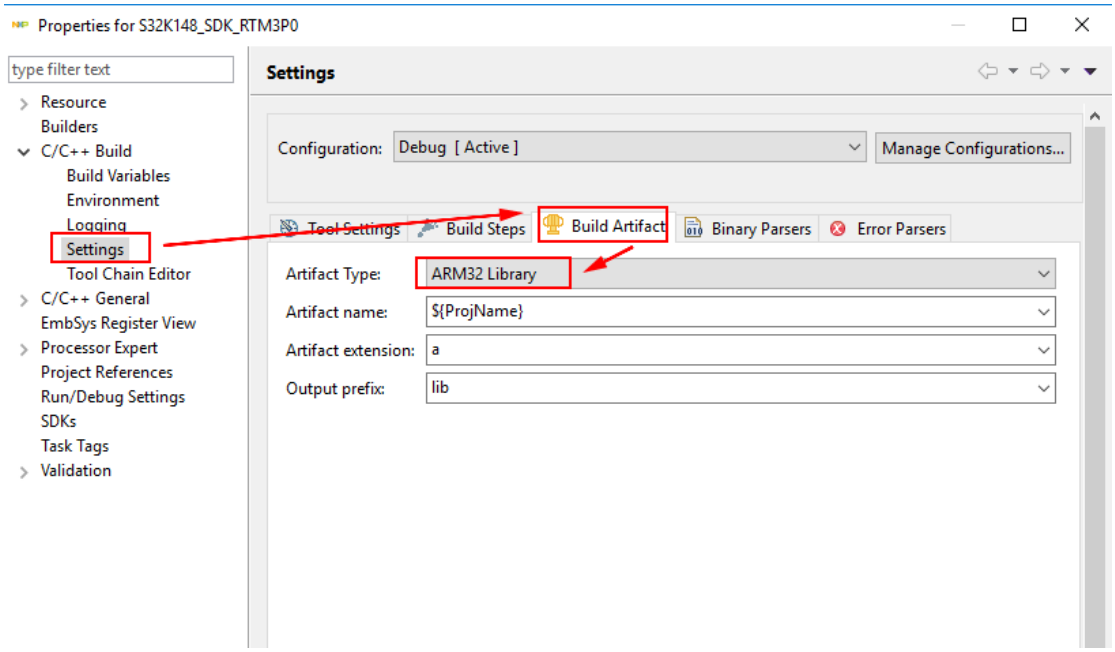


Figure 38. setting the project build target as a library

(4). Delete or exclude the file--**main.c, LIN21.ldf, startup.c, system_S2K148.c** and **startup_S32K148.S** from the project build target which can be found in the later C++ application project.

(5) Now Processor Expert GUI can be used to configure and generate the MCU peripheral SDK LLD codes and configuration files as usual. If build successfully, you are able to see the <**Project_Name**>.a file is created.

2. Create the C++ application project.

- ✓ Step (1), (2) The same as above step (1), (2) except Language choose **C++**.
- ✓ (3) Include the headers we used in the C library project. Add below header files paths to **Project → Property → C/C++ build → Settings → Tool Settings → Standard S32DS C Compiler → Includes,** and do the same for **Standard S32DS C++ Compiler.**
  - "${workspace_loc:/S32K148_SDK_RTM3P0/Generated_Code}"
  - "${workspace_loc:/S32K148_SDK_RTM3P0/utilities}"
- ✓ (4) Include the SDK library file generated in the previous C library project. Click **Project → Property → C/C++ build → Settings → Tool Settings → Standard S32DS C++ Linker → Libraries**. Fill the following information respectively to connect with the library project..
- ➢ **Libraries(-l)**: :libS32K148_SDK_RTM3P0.a
- ➢ **Library search path(-L)**: "${workspace_loc:/S32K148_SDK_RTM3P0/Debug}"

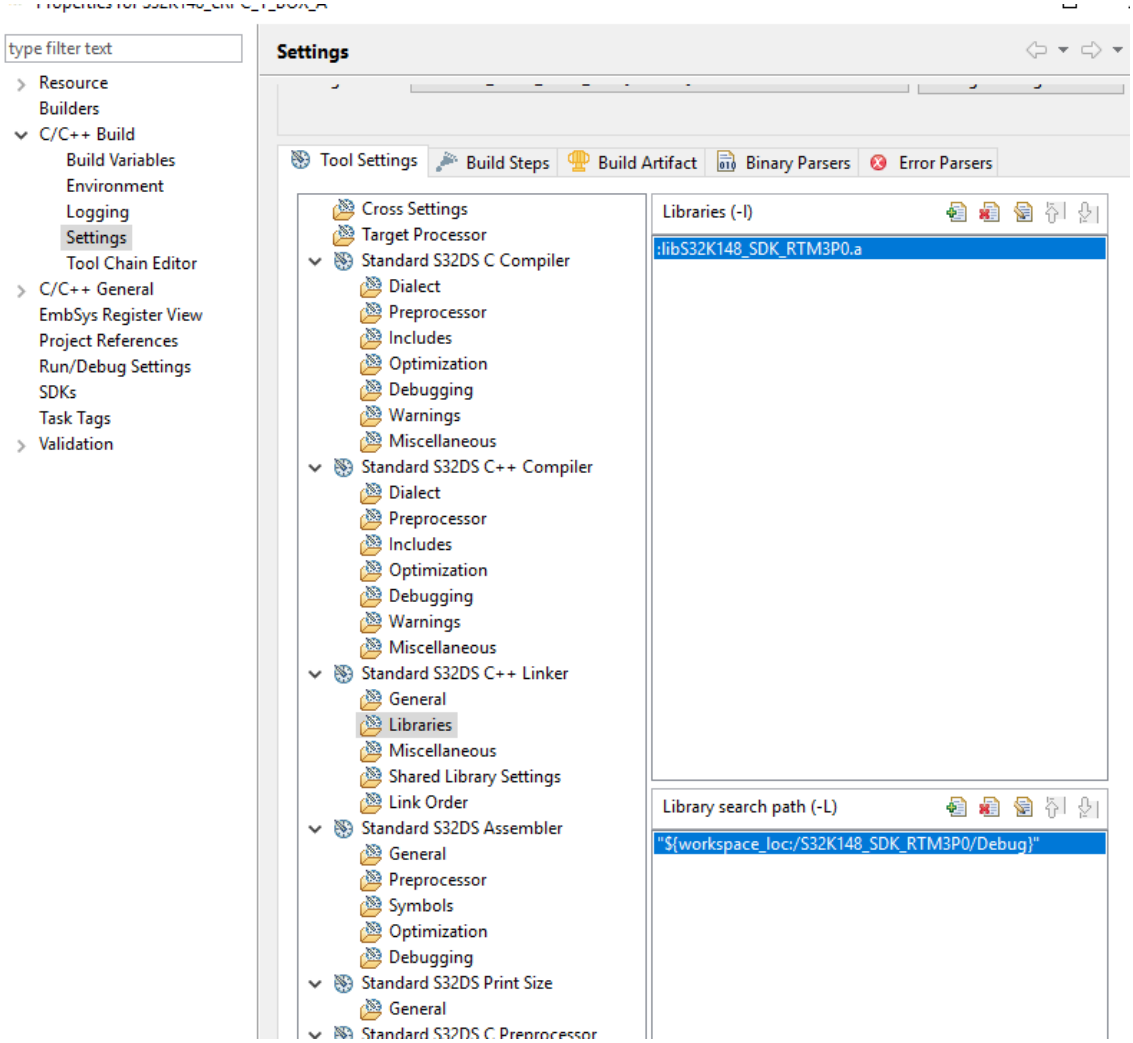


Figure 39. including the SDK library to the C++ application project

✓ (5) Enable the hardware FPU in the C++ application project to match with the C library project settings. Since S32K1xx SDK FreeRTOS porting integrated in the C library project requires hardware FPU by default, it can be configured as: Click Project → Property → C/C++ Build → Settings → Target Processor Float ABI → FP instructions(hard).

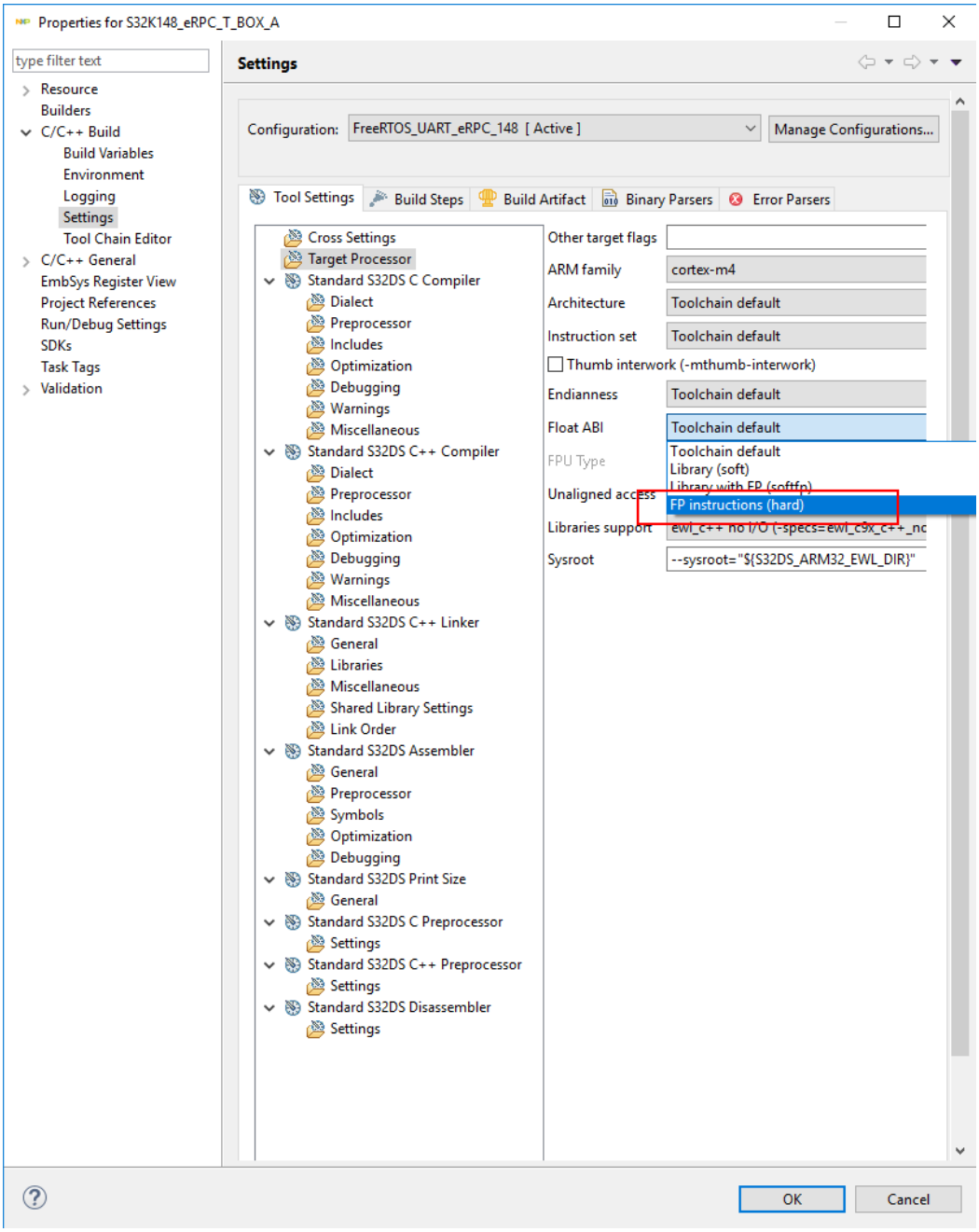


Figure 40. enabling the hardware FPU in S32DS

✓ (6) If an MCU peripheral is enabled and its interrupts are used as well in the C library project, you need to delete the relevant ISR handler weak definition in the startup files (e.g. startup_S32K148.S) in the C++ application project. Otherwise, once an enabled peripheral's IRQ occurred, CPU core will be stuck in Default ISR. The weak symbol definition will be overridden by a strong symbol definition only when they are in the same project. But in our case, the weak symbol definition is in the C++ application project, the strong symbol definition is defined in the SDK which is in another C library project. Although it's included by the C++ application project, it won't work for the strong definition to override the weak definition in another project.

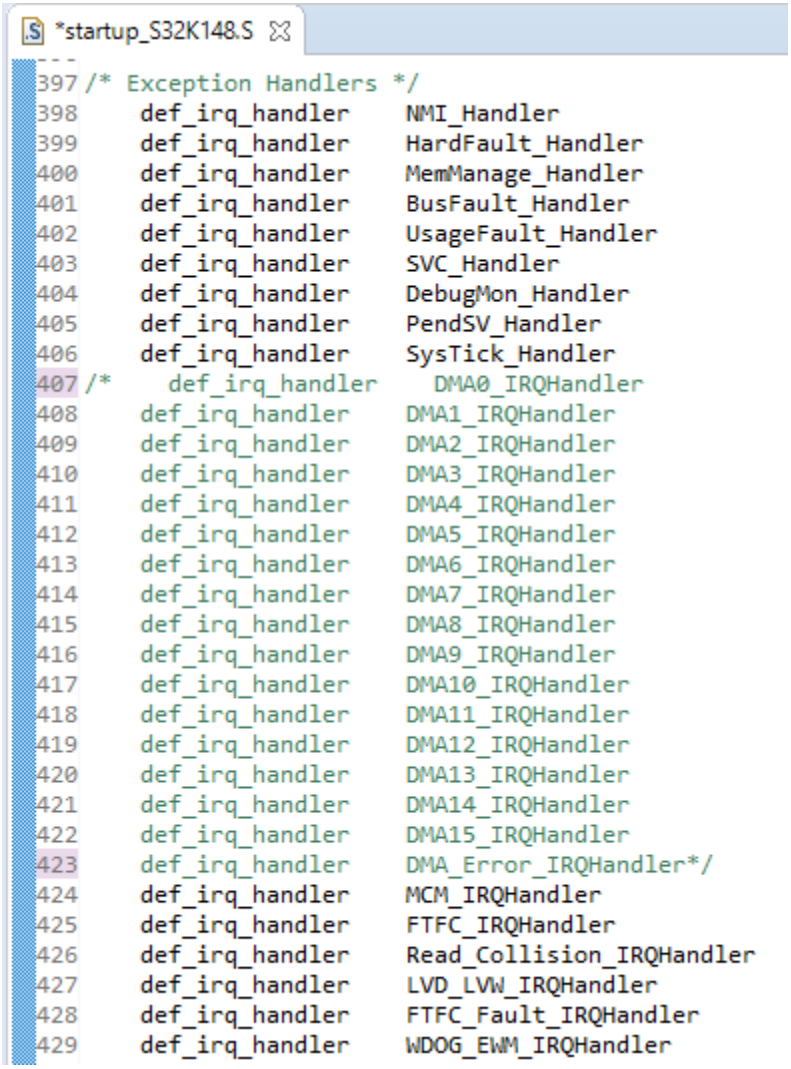


Figure 41. deleting the relevant peripheral ISR weak definition in S32DS

✓ (7) Finally, now you can start coding and have fun.

Note: A clean project action is always recommended before building the project.

3. MACRO and including path Adding

When adding a project global MACRO definition or header file include path, both C and C++ compiler options of the build target should be added, since the eRPC projects is C and C++ mixed programming.

4. About the system start order

When using UART as the transport layer, please start the server board before starting the client board. We expect the server receiving to be ready before the client sending out data. While using SPI as the transport layer, start the client board first. Because the client board is defined as SPI slave but only SPI master can start the transmission, so we used a GPIO pin

toggling to let the server inform the client to start the transmission. If the server start first, the client may miss the signal.

5. No error hint for redefinition

Since we are developing in a C++ project instead of C, if there is any variable or function redefinition, C++ will take it as overloading and there won't be error or warning hint.

6. When creating a queue in FreeRTOS, Including the header file queue.h Compiling failed.

From the error list, we found that there is a <queue.h> in C++ too, which is not the header file we want to use for FreeRTOS. So please add a specific project related path for this header file:

#include"..\..\S32K144_SDK_RTM3P0\SDK\rtos\FreeRTOS_S32K\Source\include\queue.h"

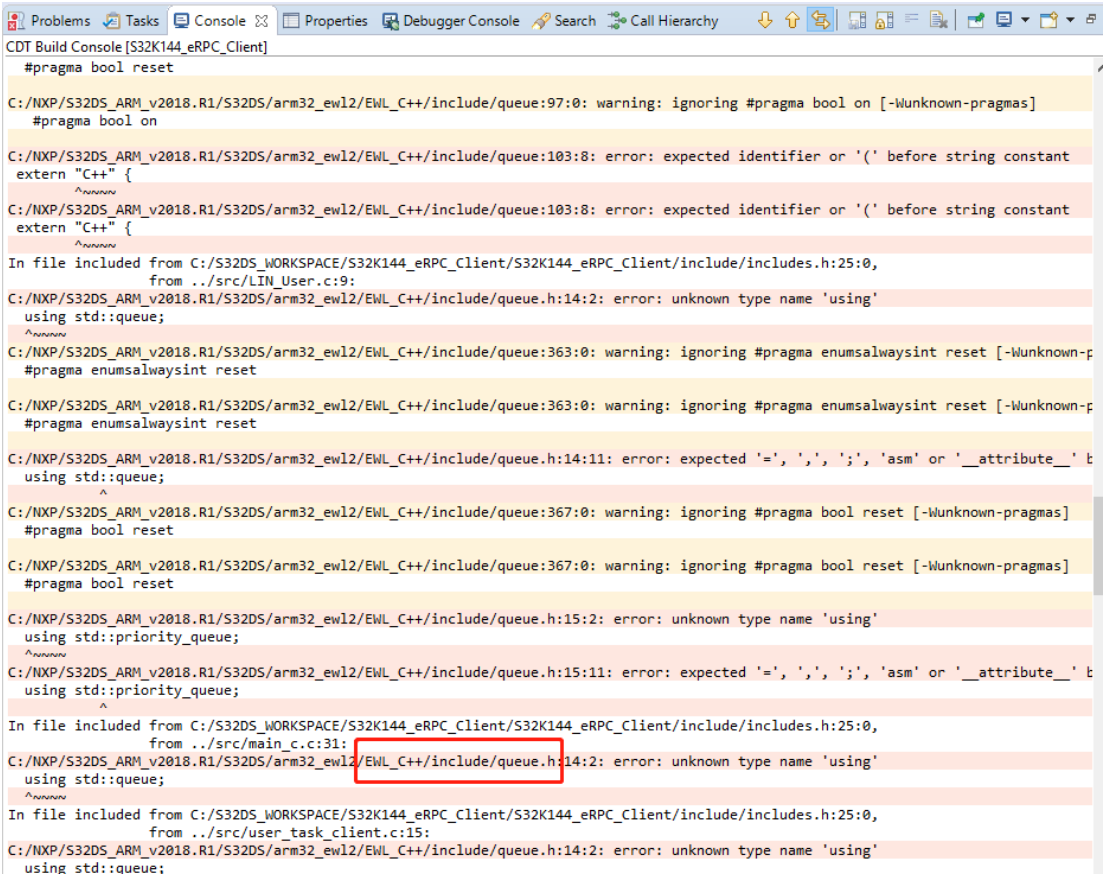


Figure 42. queue.h conflicting between FreeRTOS and C++ library

7. After adding FreeRTOS, run out of the RAM size. How to increase?

The default project linker file divides the SRAM memory into upper region and lower region. Most data in code are stored in upper section while most space in lower section is unused. We can increase the upper section and decrease lower section.

Before:

S32K1xx_flash.ld

```
/* Specify the memory areas */
MEMORY
{
  /* Flash */
  m_interrupts          (RX)  : ORIGIN = 0x00000000, LENGTH = 0x00000400
  m_flash_config        (RX)  : ORIGIN = 0x00000400, LENGTH = 0x00000010
  m_text                (RX)  : ORIGIN = 0x00000410, LENGTH = 0x0007FBF0

  /* SRAM_L */
  m_data                (RW)  : ORIGIN = 0x1FFF8000, LENGTH = 0x00008000

  /* SRAM_U */
  m_data_2              (RW)  : ORIGIN = 0x20000000, LENGTH = 0x00007000
}
```

Figure 43. default RAM memory distribution

After:

S32K144_64_flash.ld

```
/* If symbol __flash_vector_table__=1 is defined at link time
 * the interrupt vector will not be copied to RAM.
 * Warning: Using the interrupt vector from Flash will not allow
 * INT_SYS_InstallHandler because the section is Read Only.
 */
M_VECTOR_RAM_SIZE = DEFINED(__flash_vector_table__) ? 0x0 : 0x0400;

/* Specify the memory areas */
MEMORY
{
  /* Flash */
  m_interrupts          (RX)  : ORIGIN = 0x00000000, LENGTH = 0x00000400
  m_flash_config        (RX)  : ORIGIN = 0x00000400, LENGTH = 0x00000010
  m_text                (RX)  : ORIGIN = 0x00000410, LENGTH = 0x0007FBF0

  /* SRAM_L */
  m_data                (RW)  : ORIGIN = 0x1FFF8000, LENGTH = 0x00001000

  /* SRAM_U */
  m_data_2              (RW)  : ORIGIN = 0x1FFF9000, LENGTH = 0x0000E000
}
```

Figure 44. modified RAM memory distribution

8. Is it necessary to have C++ knowledge to implement eRPC protocol to my project?

It will be helpful to understand the eRPC work mechanism and when you want to do some optimization on this protocol if you have C++ knowledge. But if you don't, you are still capable to use it. The original eRPC project team has provided a set of C-wrapper functions for C programmer.

9. How to implement low-power mode with eRPC on S32K1xx family MCU?

To meet low-power mode requirement of the system with eRPC implementation, both client and server boards/MCU must have their own low-power codes to manage its own CPU core and used peripherals for low power mode entry and exit. Besides, a system level low-power mode management mechanism is needed to ensure a proper low-power mode entry/exit sequence between the two boards/MCU, for example, assign all wake-up sources on only one MCU(either the client or the server side is OK) and use an extra GPIO pin to generate a rising/falling edge signal to inform another MCU for low-power mode entry and wake up it from low-power mode.
In the sample project provided in this project, to test the low power mode, PTD4 is required to be connected between the server board (set as input) and the client board (set as output). Send a CAN frame with ID=222 to the client board, PTD4 will be set to high and both boards will go to VLPS mode. Press button SW2 or SW3 on the client board, the client board will wake up and set PTD4 low to inform the server to wake up. The measured current for both boards in VLPS mode is less than 70 uA respectively. The implementation code for low power entry and exit handle is in the hook function of the FreeRTOS idle task.

10. How to implement function safety with eRPC on S32K1xx family MCU?

As function safety is becoming a compulsive requirement for future automotive ECU application, S32K1xx family MCU is designed to target ISO 26262 ASIL-B function safety application. NXP provides S32K1xx safety manual, FMEDA and safety library/framework software under a signed NDA to help user achieve the function safety goal, details can be referred from NXP website(www.nxp.com). With the eRPC porting on S32K1xx, user can add application function based critical data check/verify and/or redundant peripheral check to help detect random and systematic failures between the two S32K1xx MCU to achieve a higher system function safety goal.

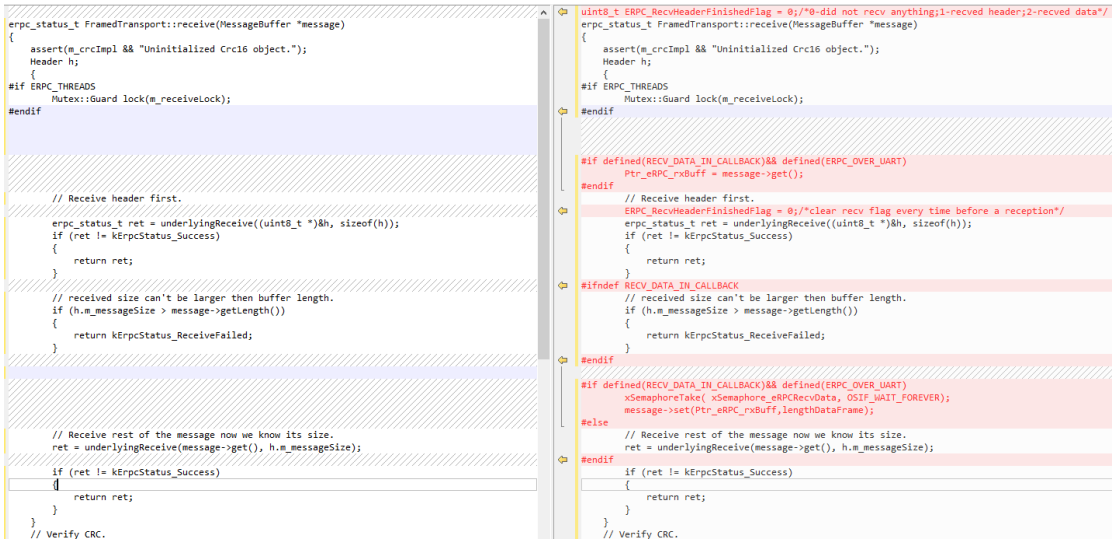
Appendix C. Improvement on original eRPC protocol

1. Using hardware CRC

Using S32K1xx MCU hardware CRC instead of software CRC to reduce CRC calculation time, **10x** faster can be improved. There are 4 times of CRC calculation will be executed for a completed eRPC calling period if it's not an oneway message type.

2. Optimizing UART RX receiving efficiency.

Calling **UART_SetRxBuffer()** in UART RX callback instead of calling **underlyingReceive**() again after receiving the eRPC header in transport layer when using USART as the eRPC sync communication interface. In such way, the following eRPC message/response frame can be received timely with reducing the receive setting up time. In its original code, there is a possibility the server can only get the 4 bytes header and loss the coming data frame because data send out before receiving is set up. The failure ratio varies based on different baud rate and other peripheral interrupts used in application according to our test. SPI doesn't have such an issue since SPI used another GPIO to inform the sender (SPI master) after the receiver (SPI slave) is set up receiving. Similar methods can be implemented on UART, or alternatively using LPUART's hardware flow control.



```
erpc_status_t FramedTransport::receive(MessageBuffer *message)
{
    assert(m_crcImpl && "Uninitialized Crc16 object.");
    Header h;
    {
#if ERPC_THREADS
        Mutex::Guard lock(m_receiveLock);
#endif

        // Receive header first.
        erpc_status_t ret = underlyingReceive((uint8_t *)&h, sizeof(h));
        if (ret != kErpcStatus_Success)
        {
            return ret;
        }
        // received size can't be larger then buffer length.
        if (h.m_messageSize > message->getLength())
        {
            return kErpcStatus_ReceiveFailed;
        }

        // Receive rest of the message now we know its size.
        ret = underlyingReceive(message->get(), h.m_messageSize);
        if (ret != kErpcStatus_Success)
        {
            return ret;
        }
    }
    // Verify CRC.
```

```
uint8_t ERPC_RecvHeaderFinishedFlag = 0;/*0-did not recv anything;1-recved header;2-recved data*/
erpc_status_t FramedTransport::receive(MessageBuffer *message)
{
    assert(m_crcImpl && "Uninitialized Crc16 object.");
    Header h;
    {
#if ERPC_THREADS
        Mutex::Guard lock(m_receiveLock);
#endif

#if defined(RECV_DATA_IN_CALLBACK)&& defined(ERPC_OVER_UART)
        Ptr_eRPC_rxBuff = message->get();
#endif
        // Receive header first.
        ERPC_RecvHeaderFinishedFlag = 0;/*clear recv flag every time before a reception*/
        erpc_status_t ret = underlyingReceive((uint8_t *)&h, sizeof(h));
        if (ret != kErpcStatus_Success)
        {
            return ret;
        }
#ifndef RECV_DATA_IN_CALLBACK
        // received size can't be larger then buffer length.
        if (h.m_messageSize > message->getLength())
        {
            return kErpcStatus_ReceiveFailed;
        }
#endif
#if defined(RECV_DATA_IN_CALLBACK)&& defined(ERPC_OVER_UART)
        xSemaphoreTake( xSemaphore_eRPCRecvData, OSIF_WAIT_FOREVER);
        message->set(Ptr_eRPC_rxBuff,lengthDataFrame);
#else
        // Receive rest of the message now we know its size.
        ret = underlyingReceive(message->get(), h.m_messageSize);
#endif
        if (ret != kErpcStatus_Success)
        {
            return ret;
        }
    }
    // Verify CRC.
```

Figure 45. the original receive function and optimized receive function

Implementation for RX callback to set up the new RX buffer to receive the eRPC data after receiving 4 bytes header.

```
void eRPC_UART_RX_Callback(void *driverState, uart_event_t event, void *userData)
{
#ifdef RECV_DATA_IN_CALLBACK
    lpuart_state_t * ptr = (lpuart_state_t*) driverState;
    BaseType_t xResult,xHigherPriorityTaskWoken = pdFALSE;

    if(UART_EVENT_RX_FULL == event)
    {
        if(0 == ERPC_RecvHeaderFinishedFlag)
        {   /*receive header finished*/
            ERPC_RecvHeaderFinishedFlag = 1;
            lengthDataFrame = (*(uint16_t*)(&ptr->rxBuff[0]));
            /*set buffer for the coming content right after received the header.*/
            UART_SetRxBuffer(&uart_pal1_instance,Ptr_eRPC_rxBuff,lengthDataFrame);
        }
        else
        {   /*receive the content finished.*/
            xResult = xSemaphoreGiveFromISR( xSemaphore_eRPCRecvData, &xHigherPriorityTaskWoken );
            ERPC_RecvHeaderFinishedFlag = 2;
        }
        /* Was the message posted successfully? */
        if( xResult != pdFAIL )
        {
            /* If xHigherPriorityTaskWoken is now set to pdTRUE then a context
            switch should be requested.  The macro used is port specific and will
            be either portYIELD_FROM_ISR() or portEND_SWITCHING_ISR() - refer to
            the documentation page for the port being used. */
            portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
        }
    } « end if UART_EVENT_RX_FULL==e... »
#endif
} « end eRPC_UART_RX_Callback »
```

Figure 46. set up the new RX buffer in the UART RX callback

3. Using Hardware flow control for synchronization.

As described earlier, hardware flow control is used to ensure data send before the receiver fully set up. Alternatively, user can use another IO to do the synchronization, which can be compatible with FlexIO UART. But in this way extra CPU source is needed and one IO interrupt is used. So, hardware flow control is recommended for eRPC. Below figure shows the steps:

1. Choose the GPIO used for flow control and re-generate code.

Signals	Pin/Signal Selection
▾ LPUART0	
Clear to Send	PTA0
Request to Send	PTA1
Receive Data	PTA2
Transmit Data	PTA3

Figure 47. pins definition of UART hardware flow control

2. Connect CTS on board A to RTS on board B, connect CTS on board B to RTS on board A.
3. Set below registers to enable the function after initializing USART.

```
/*setting UART hard flow control start*/
LPUART0->MODIR |= LPUART_MODIR_TXCTSE(1); /*enable CTS*/
LPUART0->MODIR |= LPUART_MODIR_RXRTSE(1); /*enable RX RTS*/
```

4. Now you can see from the logic analyzer that each time before the sender board sending data, RTS on the receiver board will set the signal high. When set to low, the sender will hold the data until get high again.

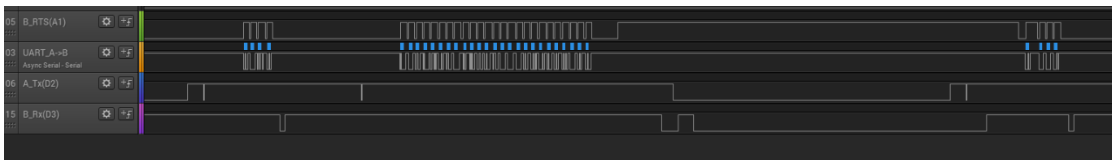


Figure 48. UART hardware flow control waveform

4. SPI slave transfer completion issue

When using SPI as the transport layer, the server is defined as SPI slave, the client is defined as SPI master. The transfer type is interrupts. When the server finishes receiving data, a flag is expected to be set from the SPI slave ISR callback. Then the server will get the flag and exit the receiving function. In the test, we found that there is a chance the flag is set before 4 bytes header reception is finished. Then the code after will fail and eRPC process will exit in advance. There are 2 ways to solve this problem, one is to change transfer type to DMA instead of interrupts, second is to use another IO pin to set in the client board SPI master callback and set a flag in the server board IO ISR.

Appendix D. Reference

1. S32K1xx MCU Family - Data Sheet(REV 13), www.nxp.com, 16 July, 2019.

2. S32K1xx MCU Family - Reference Manual(REV 13), www.nxp.com, 16 July, 2020.

3. S32K148 Automotive Telematics Box (T-Box) solution,

https://www.nxp.com/S32K148-T-BOX, December, 2018.

4. eRPC GitHub repository：https://github.com/EmbeddedRPC/erpc/

Appendix E. Abbreviations Used in the Document

BCM	**B**ody **C**ontrol **M**odule
DCU	**D**omain **C**ontrol **U**nit
RPC	**R**emote **P**rocedure **C**alls
eRPC	**E**mbed **R**emote **P**rocedure **C**alls
IDL	**I**nterface **D**efinition **L**anguage
CRC	**C**yclic **R**edundancy **C**heck
SDK	**S**oftware **D**evelopment **K**it

Appendix F. Reversion History

Reversion	Date	Description of changes
Rev 0.1	7/2020	Initial release for internal review.
Rev 0.2	8/2020	Add low-power mode and function safety implementation with eRPC.
Rev 1.0	9/2020	Revised for initial release.