life.augmented

AN5413

Application note

Getting started with the SPC58x Networking

Introduction

SPC58x Automotive micro-controllers features 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.

This application note presents a demonstration package built on top of the FreeRTOS+TCP/IP stack that contains some applications built on top of it to test basic and complex networking features and measuring some Ethernet controller performances.

All application demos have been developed by using SPC5Studio tool https://www.st.com/en/development-tools/spc5-studio.html.

SPC5Studio is built-on Eclipse that provides functionalities and components to set up both hardware and software stacks on STMicroelectronics automotive micro-controllers.

SPC5Studio also offers a simple approach to set up signals and clocks mandatory to configure the hardware.

When testing an Ethernet communication interface, the TCP/IP stack is necessary for communicating over a local or a wide area network.

The application note also introduces the main Ethernet controller features.

It covers the Ethernet and physical layer programming guide.

All the information available in this document can be considered and adopted, with minor variances, for all SPC58x MCU family where the Ethernet controllers are embedded.

1 Ethernet overview

The 10/100 Mbit/s Ethernet provides many advanced features that can be summarized into the following categories:

- MAC Core
- MAC Transaction Layer (MTL)
- DMA Block
- SMA Interface
- MAC management counters
- Power Management block (PMT)

1.1 Core Interface

MAC core implements both main transmission and reception features as well as advanced standards.

1.2 MAC Transaction Layer

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

The main components of the Core/MTL layers are:

- CSMA/CD Protocol support
- Energy Efficient Ethernet
- RMON Counters
- IEEE 1588-2002 and IEEE 1588-2008 timestamping
- Two VLAN tag stripping
- Wake Up capability
- RX/TX Multiple queues
- TBI, SGMII PHY interfaces (PCS)
- Transmission
  - SFD insertion, CRC padding, Jumbo frames, Store-and-Forward mechanism or threshold mode, collision handling.
  - IPv4 header checksum and TCP, UDP, or ICMP checksum insertion.
- Receive
  - Flexible address filtering modes
  - Additional filters for L3/L4, VLAN tagging
  - Layer 3/Layer 4 checksum offloading
  - Source Address filtering
  - TX VLAN TAG insertion/replacement
  - Auto FCS strip

1.3 DMA engine

The Direct Memory Access is an advanced DMA block inside the Ethernet controller designed to exchange data between the internal FIFOs and system memory.

The engine provides interrupt to control, transmit and receive packets.

1.3.1 DMA descriptor overview

Transfer of Ethernet packets between Transmit/Receive FIFOs and memory is performed by direct memory access (DMA) using transfer descriptors.

An Ethernet packet can span over one or multiple DMA descriptors. One DMA descriptor can be used for one Ethernet packet only.

The last descriptor points to the first one for forming a ring of descriptors.

The Interrupt generation on Frame Transmit and Receive transfer completion is supported and the status is reported inside the descriptor itself.

The controller supports the following two types of descriptors:

- Normal Descriptor: used for packet data and to provide control information applicable to the packets to be transmitted or received.
- Context Descriptor: used to control packet advanced features, e.g.: timestamps, VLAN.

This application covers the normal descriptors programming.

Both Transmit and Receive Normal descriptors have the following two formats:

- Read format
- Write-Back format.

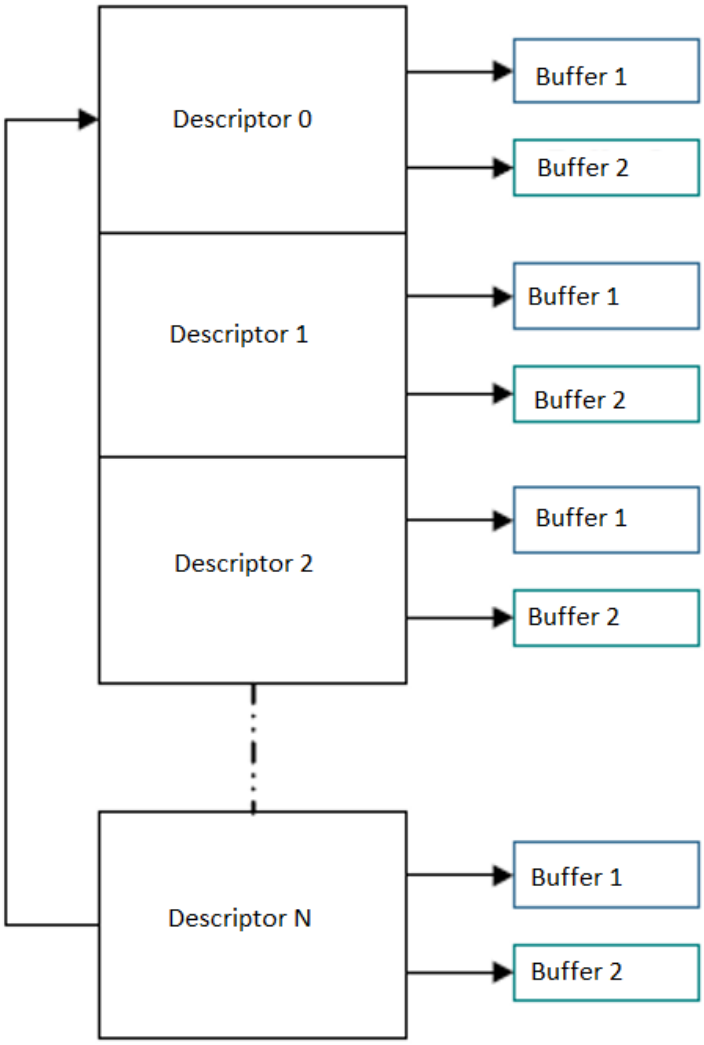
When the descriptor is manipulated by the software driver, for example, to transmit a packet, it is in Read Format.

After transmitting the packet, the latest descriptor (that has been owned by the DMA engine) results in Write-Back format and it contains the status of the packet transmission.

On the reception side, the software driver prepares the descriptors in read format, as soon as a frame is received, the descriptor is in write back format and it contains the status of incoming frame so handled by the DMA engine.

Refer to the "Descriptor" related chapter inside the Reference Manual of the micro-controller for all the details and related programming guide (see Section Appendix B Reference documents) .

Figure 1. Descriptor ring



1.4 SMA Interface

The Ethernet can access the PHY registers through the Station Management Agent (SMA) module.

This allows to access PHY registers from internal Ethernet ones implementing the Management Data Input/ Output (MDIO) serial bus.

The media-independent interface (MII) is a standard used to connect an Ethernet block to a PHY:

- MII - Media independent interface;
- RMII - Reduced media independent interface.

Please refer to the TN1305 Technical note (Section Appendix B Reference documents) Network Management Interfaces.

1.5 MAC Management counters

The MAC Management Counters (MMC) module is to enable various counters according to the Remote Monitoring (RMON) standard specification (RFC2819/RFC2665). The low-level driver logs the following counters inside the private structure:

- Generic Counters Parameters
- Transmit Counters Parameters
- Receive Counters Parameters
- IPC Receive Counters Parameters

2 Application set up

According to the PCB design and MII mode, the connection between Ethernet and PHY transceiver can be complex, so SPC5Studio provides a set of graphical interfaces to configure the signals, clocks and so on.

2.1 Signal configurations

SPC5Studio allows to configure the signals to connect the Ethernet to the Physical device, i.e.: the MDIO bus and, according to the MII mode supported by the platform, it also guarantees the right programming of all the MII signals, e.g: RXDATA, TXDATA, TX_CLK, RX_CLK and so on.

The two figures below show both the graphical interface available to assign the signals and a table where PIN capabilities can be tuned according to the PCB requirements.

Figure 2. Pinmap editor table

SPC58xGxx Board Initialization Component RLA

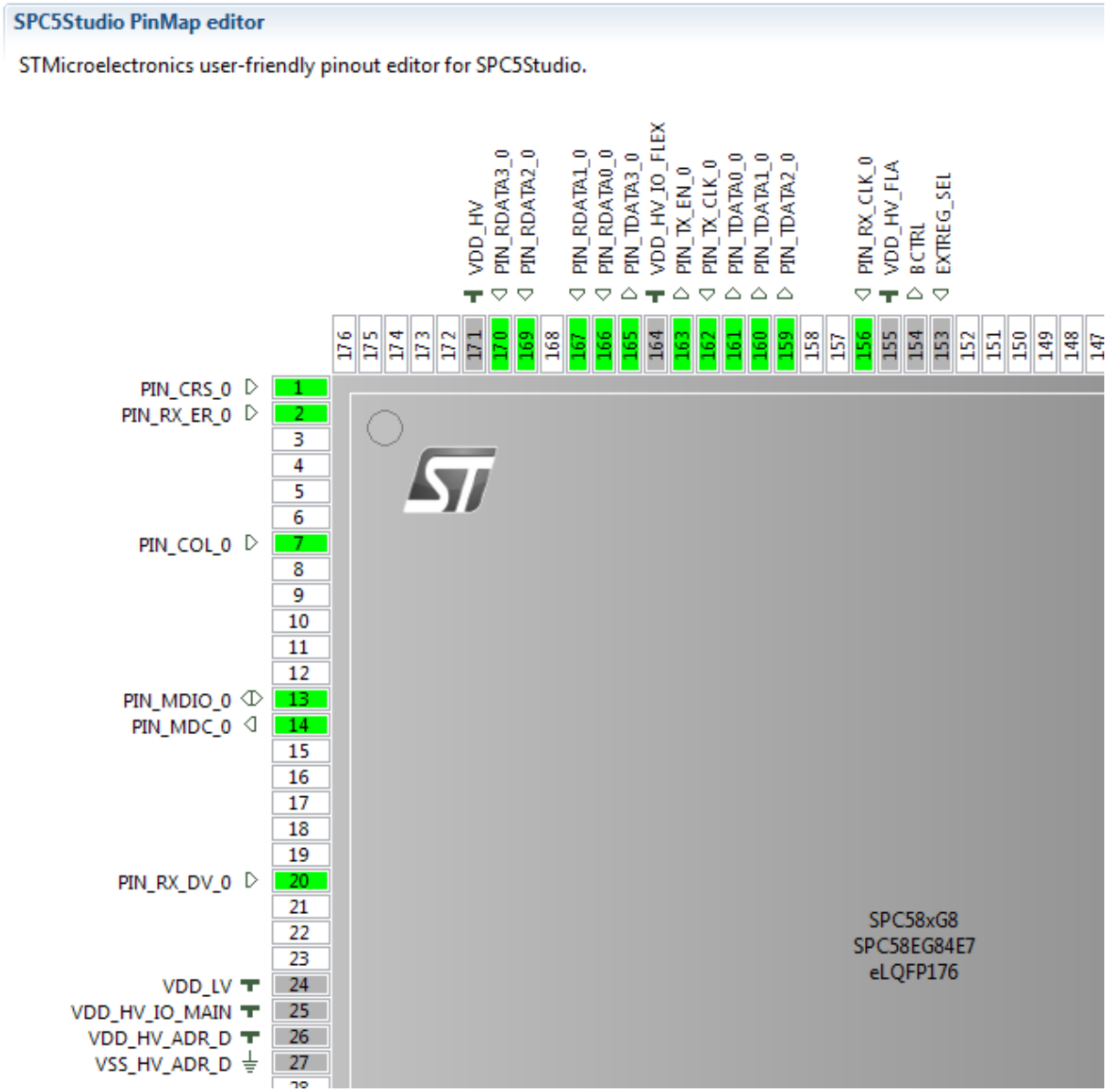
Board options and settings.

Pins List

#	Identifier	Port	Bit	Pin Mode	Latched State	MSCR Index	SSS	MSCR Index	SSS
0	PA_LED1	PORT_A	2	MODE_IO	LOW	2	0	512	0
1	PIN_MDIO_0	PORT_C	0	MODE_IO	LOW	32	13	924	8
2	PIN_RDATA1_0	PORT_C	10	MODE_INPUT	LOW	42	0	928	4
3	PIN_RDATA0_0	PORT_C	11	MODE_INPUT	LOW	43	0	927	3
4	PIN_TDATA3_0	PORT_C	12	MODE_OUTPUT	LOW	44	14	512	0
5	PIN_TX_EN_0	PORT_C	13	MODE_OUTPUT	LOW	45	16	512	0
6	PIN_TX_CLK_0	PORT_C	14	MODE_INPUT	LOW	46	0	923	3
7	PIN_TDATA0_0	PORT_C	15	MODE_OUTPUT	LOW	47	1	512	0
8	PIN_COL_0	PORT_C	5	MODE_INPUT	LOW	37	0	931	2
9	PIN_TDATA2_0	PORT_D	0	MODE_OUTPUT	LOW	48	5	512	0

Note: In the SIUL2 configuration the Output Edge Rate Control (OERC) bit for output pins are set as Strong / Very Strong drive.

Figure 3. **Pinmap wizard (Ethernet pins)**



2.2 Clock settings

SPC5Studio configures by default all the MCU clocks according to the specifications for the micro-controller.

The Ethernet host clock comes from the PBRIDGE_CLK source.

However the Ethernet controller can require other clocks for the MII interface.

These clocks can be provided in different ways according to the selected MII mode. More details are available in the Reference Manual for a Micro-Controller (Section Appendix B Reference documents).

For example, in case of MII (default on SPC58x reference boards) the RXCLK and TXCLK come from the PHY device to the Ethernet controller. These are input signals.

A common issue on this Controller is that the application stuck on DMA software resets if the TXCLK is not properly configured.

3 FreeRTOS overview

FreeRTOS TCP/IP component, integrated inside SPC5Studio, provides a scalable, open source and thread safe TCP/IP stack for FreeRTOS.

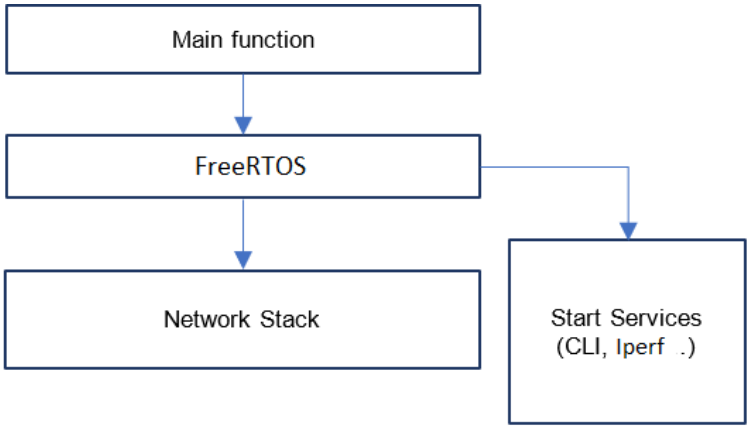
SPC5Studio provides, in the pre-built applications, a main code that, as detailed in the diagram below is based on FreeRTOS Operating System.

The **main** function starts the OS scheduler that is delegated to initialize the whole network processes:

- TCP/IP stack;
- Ethernet interface;
- Physical layer.

Other tasks can be also scheduled to start services: e.g. CLI, HTTP and FTP servers.

Figure 4. FreeRTOS flow diagram

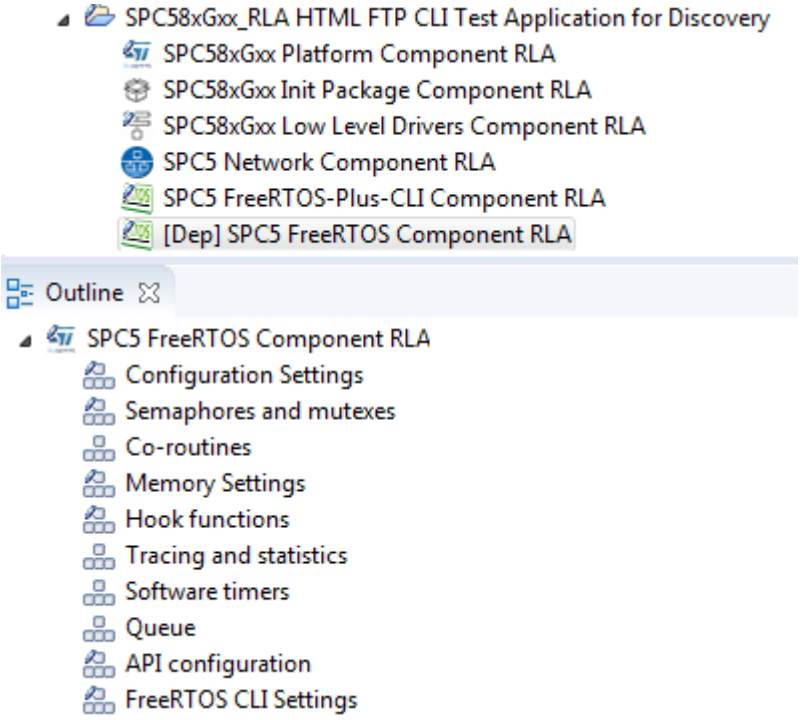


The FreeRTOS component is installed with a default configuration to cover all network needs in terms of task priorities and features.

Expert User can tune some parameters for memory management, synchronization, timers etc.

This can be done by graphical interface as shown below; all selections are documented in the Help section available in the tool.

Figure 5. **SPC5Studio FreeRTOS outline view**



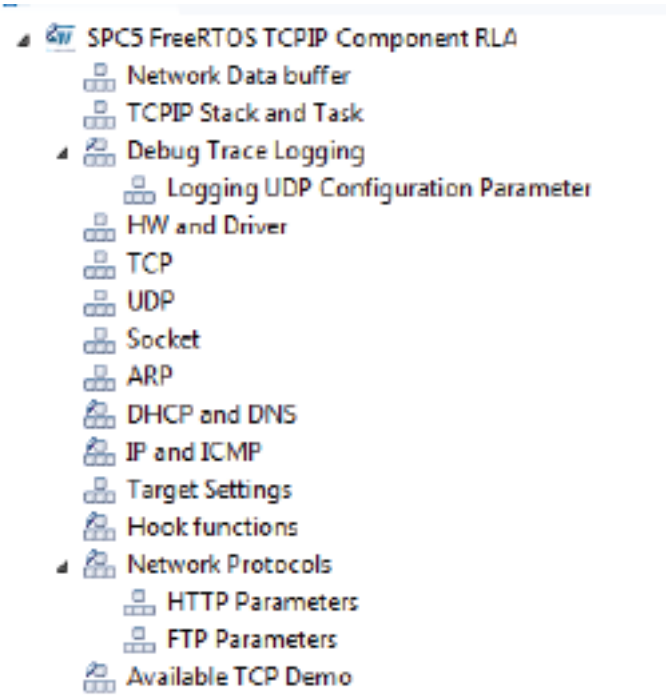
4 FreeRTOS TCP/IP configuration

The FreeRTOS TCP/IP component provides a configuration file *FreeRTOSIPConfig.h* which defines all the configuration parameters, it is automatically generated by SPC5Studio depending on the values set on FreeRTOS TCP/IP component graphical interface.

Below the Outline of the FreeRTOS TCP/IP component showing the main groups of configuration parameters.

Note: *Note that the FreeRTOSIPConfig.h must not be manually modified. Use the FreeRTOS TCP/IP graphical configuration tool instead.*

Figure 6. TCP/IP parameter view

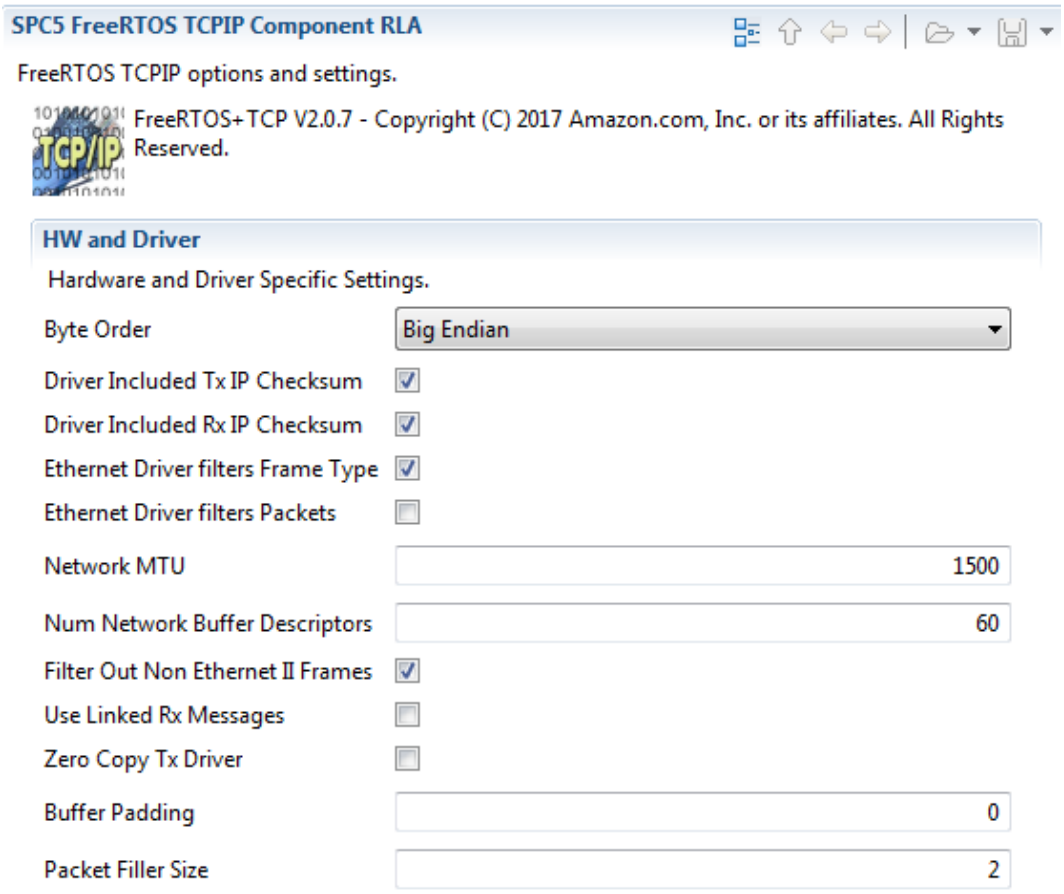


The TCP/IP stack can be complex to tune and configure, so the application provides a default set up suitable to run the whole stack. Expert User can change the default parameters for TCP / UDP and Socket. It is possible to enable the loopback mode and configure the echo reply on ping. It is also possible to install user hook.

The figure below shows the “HW and Driver Specific Settings” configuration, by default the full checksum computation is performed by the Ethernet controller.

Note: *in this application the standards MTU are used by default.*

Figure 7. **TCP/IP component view**



Different buffer allocation schemes suite different embedded applications, so FreeRTOS+TCP keeps the buffer allocation schemes as parts of the TCP/IP stack's portable layer.

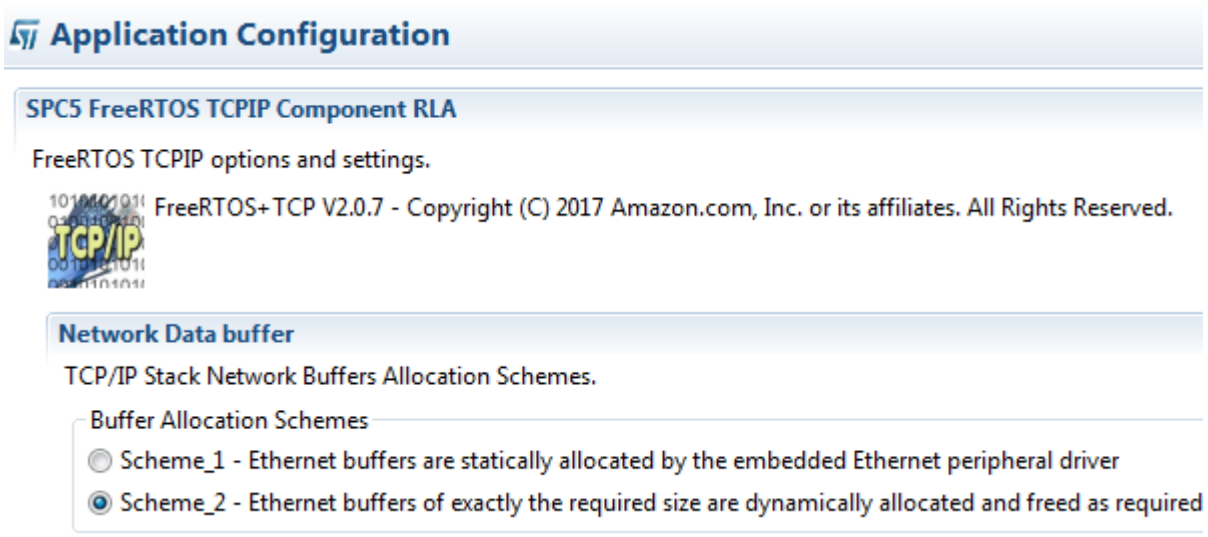
At the time of writing, two example buffer allocation schemes are provided - each with different tradeoffs among simplicity, RAM usage efficiency, and performance.

The C source files that implement the buffer allocation schemes are located in portable/BufferManagement directory.

- Scheme 1 – BufferAllocation_1.c
  - Ethernet buffers are statically allocated by the embedded Ethernet peripheral driver (at compile time). This ensures the buffers are aligned as required by the specific Ethernet hardware. This is done by means of the vNetworkInterfaceAllocateRAMToBuffers that must be implemented in the portable layer if the user wants to use this scheme.
- Scheme 2 – BufferAllocation_2.c
  - Ethernet buffers of exactly the required size are dynamically allocated and freed as required. This requires a fast memory allocation scheme that does not suffer from fragmentation.

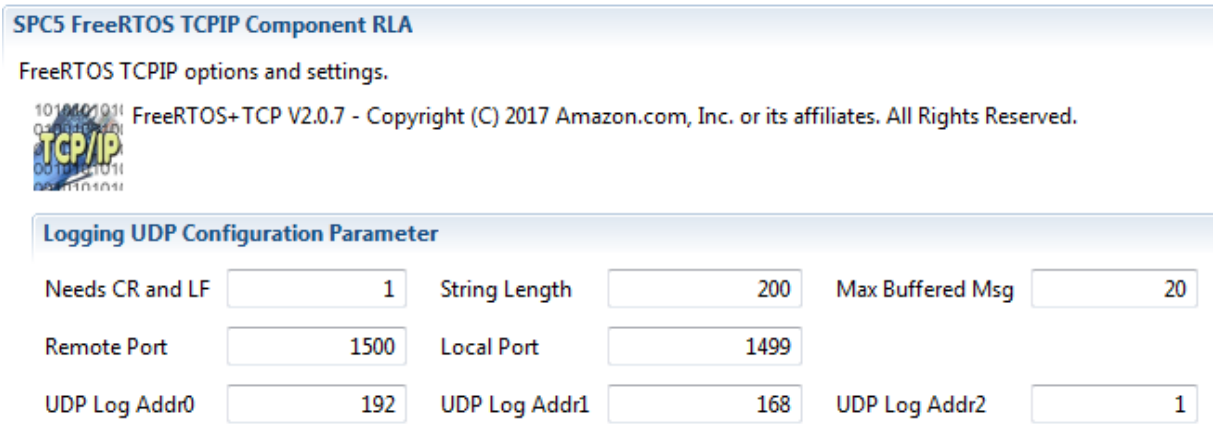
In SPC5Studio the FreeRTOS TCP/IP Component is configured to use BufferAllocation_2 as showed in the following figure:

Figure 8. **TCP/IP buffer allocation schema**



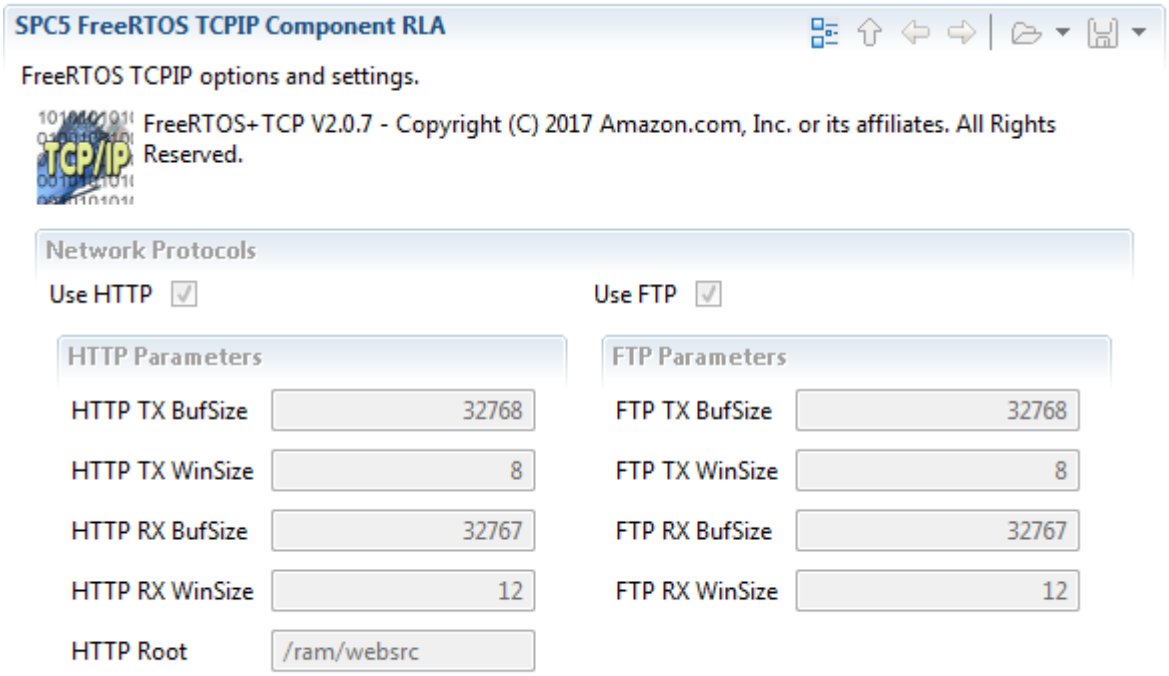
Some features can be also turned-on or off, for example the Logging print messages through UDP frames. The figure below shows how to configure the server and UDP ports for this support.

Figure 9. **UDP logging window**



HTML and FTP servers can be enabled and users callback can be set up to debug monitor conditions. The figure below shows the default configuration for HTML an FTP servers.

Figure 10. **HTTP and FTP server configurations**



5 Network component configuration

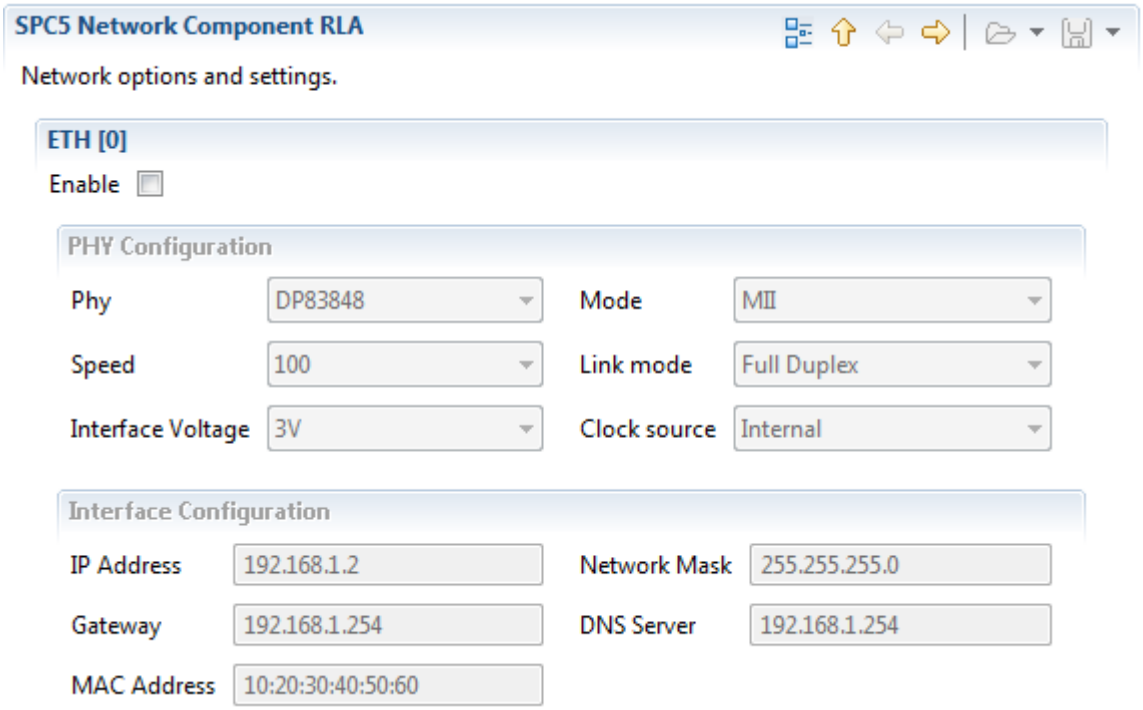
The SPC5Studio Network component is used to configure both Ethernet and PHY devices.

The graphical interface offers several options to configure the devices.

For the selected physical driver, it is possible to change the MII mode, fix the speed and duplex mode (skipping auto-negotiation process of the link).

For the Ethernet interface, as shown in the figure below, it is possible to provide a different address and setting either static IP or DHCP address.

Figure 11. Ethernet configuration inside the Network component view



6 FreeRTOS_IPInit

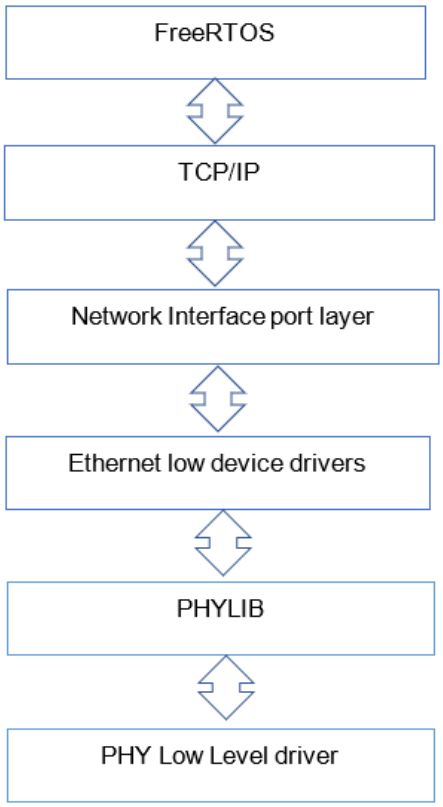
The initialization of the whole TCP/IP Stack is done through the networkInit() function, that is called by the componentInit() helper.

The FreeRTOS_IPInit() function, receives as input parameters the Ethernet configurations settings from the graphical interface (please refer to Network Component in the next chapters).

It creates a prvIPTask task that configures and initializes the network interface by invoking the xNetworkInterfaceInitialise() function.

The following diagram shows a simplified diagram of the layers in the network stack

Figure 12. Generic Network flow diagram



7 Network Interface Port Layer

The network interface port layer provides an interface between the embedded TCP/IP stack and the Ethernet driver; it is implemented inside the NetworkInterface.c file.

The network interface port layer must provide the following main routines:

- Section 7.1 xNetworkInterfaceInitialize to initialize the Ethernet and PHY drivers.
- Section 7.2 xNetworkInterfaceOutput to implement the packet transmission.
- Section 7.3 prvNetworkInterfaceInput to implement the reception of a packet.

The initialization process is based on a defined type structure that represents the network driver operations: this is the netdrv_ops_t and defined inside the NETDRV_IF header file.

7.1 xNetworkInterfaceInitialize

When FreeRTOS TCP task is scheduled, it invokes the `xNetworkInterfaceInitialise` to set up the Ethernet controller allocating all the resources and establishing the Link.

In summary, the `xNetworkInterfaceInitialise` will do the following steps:

- Initialize the function callbacks and private structure according to the user configuration.
- Configure the Ethernet parameters in the micro-controller specific registers.
- Detect the PHY and start-up the Link.
  - xNetworkInterfaceInitialise fails until the link is not established.
- Configure the descriptor rings and allocate the buffers.
- Initialize the DMA and perform its software reset.
- Initialize the CORE/MTL and internal counters.
- Invoke the xTaskCreate to create the main task
  - It is invoked by the interrupt service routine to manage the incoming frame receptions and transmit completion.
- Start the reception and transmission processes and configure the related tail pointer registers.

```
BaseType_t xNetworkInterfaceInitialise( void ){
    NetworkDriver *netd;
    uint32_t ret;
    netd = &ETHD0;
    netd->netdrv.instance = SPC5_ETH0_INSTANCE;
    spc5_eth0_netdrv_init(&netd->netdrv);

    if( netd->net_task_handle != NULL ) {
        return pdFAIL;
    }
    /* Initialize the PHY driver */
    netd->phydrv = phy_init( &netd->netdrv );
    if( netd->phydrv == NULL ) {
        return pdFALSE;
    }
    /* Setup the selected PHY */
    ret = phy_setup( netd->phydrv );
    if ( ret == pdFALSE) {
        return pdFALSE;
    }
    /* Configure the MAC and exit if no link is UP */
    phy_get_link( netd->phydrv );
    /* Setup the network driver */
    netd->netdrv.setup( netd->netdrv.priv, (uint8_t *) netd->Rx_Buff, (uint8_t *) netd-
>Tx_Buff );
    /* Configure the connection between the network driver and the PHY */
    netd->netdrv.phy_config( netd->netdrv.priv, phy_get_speed( netd->phydrv ),
phy_get_link_mode( netd->phydrv ) );

    xTaskCreate( prvNetHandlerTask, "EMAC", configNET_TASK_STACK_SIZE, netd,
      NET_TASK_PRIO, &netd->net_task_handle );
    /* Set notification task to low level driver */
    netd->netdrv.set_notify( netd->netdrv.priv, (void *)netd->net_task_handle );
    /* Link is Up and HW ready to start... */
    return pdPASS;
}
```

7.2 xNetworkInterfaceOutput

The xNetworkInterfaceOutput sends data received from the embedded TCP/IP stack to the Ethernet driver for transmission. The TCPI/IP stack calls this function whenever a network buffer is ready to be transmitted.

The related function prototype is:

```
BaseType_t xNetworkInterfaceOutput( xNetworkBufferDescriptor_t * const pxDescriptor,
     BaseType_t bReleaseAfterSend )
```

A network buffer descriptor, xNetworkBufferDescriptor_t, is used to describe a network buffer. It points to the buffers that are transferred between the TCP/IP stack and the low level drivers.

The start of the buffer is pointed by pxDescriptor->pucEthernetBuffer while the length is located by pxDescriptor->xDataLength).

If xReleaseAfterSend is pdTRUE then both the buffer and the buffer's descriptor must be released back to the TCP/IP stack by the driver code when they are no longer required.

If xReleaseAfterSend is pdFALSE then both the network buffer and the buffer's descriptor will be released by the TCP/IP stack itself (in which case the driver does not need to release them).

The xNetworkInterfaceOutput invokes the Ethernet APIs to fill the buffer to be transmitted in the descriptor list and start the controller and DMA the transmission by updating the DMA_CH_TXDESC_TAIL_POINTER register.

7.3 prvNetworkInterfaceInput

The prvNetworkInterfaceInput() function is called by the Interrupt service routine as soon as one or more packets are received. This function moves the data from the DMA to the network buffers that have to be passed to the upper layer. This is only done if the incoming packets have no error.

The prvNetworkInterfaceInput() has to re-fill the descriptors used by the DMA engine. So the descriptors, found in write-back mode, have to be reset in normal read format mode in order to be reusable by the DMA engine.

The network interface port layer must send packets, received from the Ethernet driver to the TCP/IP stack.

7.4 ISR and DWMACHandler Task

The main Interrupt service routine is invoked when either core or DMA event occur. This is shared among Ethernet interfaces available in the MCU. The handler calls the dwmac_core_dma_main_isr function that invokes the controller low level callbacks.

The code below shows the handler for an instance of the Ethernet:

```
IRQ_HANDLER(SPC5_ETH0_CORE_HANDLER) {
    IRQ_PROLOGUE();
    dwmac_core_dma_main_isr(&drv_instance[SPC5_ETH0_INSTANCE]);
    IRQ_EPILOGUE();
}
```

where:

```
static void dwmac_core_dma_main_isr( dwmac_qos_t *dwmac )
{
    /* Get the status of CORE and DMA IRQ and clear related interrupts */
    if( dwmac->mac->isr )
    {
         dwmac->mac->isr( dwmac );
    }

    /* DMA events are managed by task */
    if( dwmac->dma->isr )
    {
        uint32_t DMA_Irq_status;

        /* Save the status of DMA IRQ: to be passed to the task handler */
        DMA_Irq_status = dwmac->dma->isr( dwmac, DMA_CH0, &dwmac->dma_irq_stat );

        xTaskNotifyFromISR( dwmac->notify, DMA_Irq_status, eSetBits, NULL );
        portYIELD_FROM_ISR();
    }
}
```

In this implementation, both transmit and receive processes are managed by prvDWMACHandlerTask:

```
ret = xTaskNotifyWait( 0x0, 0xffffffffUL, &DMA_Irq_status, portMAX_DELAY );
if( ret == pdTRUE )
{
if( DMA_Irq_status & handle_rx )
{
    prvNetworkInterfaceInput( netd );
}

if( DMA_Irq_status & rx_rbu )
{
    netd->netdrv.rx_buffer_unav( netd->netdrv.priv );
}

if( DMA_Irq_status & handle_tx )
{
    netd->netdrv.complete_send( netd->netdrv.priv );
}
}
```

The netd->netdrv.complete_send is invoked by the task to notify the Ethernet frame has been successfully transmitted.

If the frame has been transmitted without any errors it releases the transmit descriptors.

Different errors can be reported by DMA engine, as detailed inside the Reference Manual (see Section Appendix B Reference documents), each event can require different actions, for example, reallocate the related buffers, sanitize the descriptor ring, report the failure and update internal statistic software counters.

8 Ethernet platform settings

Specific platform settings are needed before starting the network stack:

- Enable the Ethernet clock.
- Set the Soc Configuration Register 0 (SIUL2_SCR0) register to select the Ethernet0 Modality, MII (Media Independent Interface) or RMII (reduced Media Independent Interface).
- Select voltage (3.3V or 5V) for IO Eth0/1 ring.

```
/* Enable clock */
SPCSetPeripheralClockMode(SPC5_ETH0_PCTL, SPC5_ME_PCTL_RUN(1) | SPC5_ME_PCTL_LP(2));

/* Select MII mode (example) */
SIUL2.SCR0.B.ETHERNET_0_MODE = 1;

/* Operate at 3.3 Volt */
PMCDIG.VSIO.B.VSIO_IM = 0;
```

9 PHY low level driver

The phy library connects the Ethernet controller with the physical medium.

This library concerns itself with negotiating link parameters with the link partner on the other side of the network connection and provides a register interface to allow Ethernet driver to determine which settings were chosen.

The library provides the callbacks to program the transceiver e.g. dp83848 10/100 MII/RMII Transceiver. So the SPC5Studio provides its own low level driver for the dp83848 transceiver, this is embedded on several STMicroelectronics automotive reference boards.

9.1 PHY initialization function

During the Ethernet controller initialization, the phylib is invoked to setup the PHY transceiver. The wole initialization process cannot proceed if the PHY layer is not configured and the link is not correctly negotiated.

Below a snapshot of the code to init the PHY driver structure for the Ethernet instance 0:

```
…
    PHYDriver *phyd = NULL;

    phyd = &PHYD1;
    phyd->phy_drv = SPC5_ETH0_PHY_NAME;
    phyd->mode = SPC5_ETH0_PHY_MODE;
    phyd->link_mode = SPC5_ETH0_PHY_LINK_MODE;
    phyd->speed = SPC5_ETH0_PHY_SPEED;
    phyd->int_mode = SPC5_ETH0_PHY_INT_MODE;
    phyd->phyops.instance = SPC5_ETH0_INSTANCE;
…
    phyd->addr = SPC5_ETH0_PHY_INIT(&phyd->phyops);
```

9.2 PHY low level APIs

The following APIs have been designed to init the transceiver, to access the PHY registers and to get the link capabilities (i.e. speed and duplex mode).

```
/* Read and Write through the MDIO bus (invoking the Ethernet SMA interface */
uint32_t phy_read(void *priv, uint32_t phy_addr, uint32_t reg_addr);
void phy_write(void *priv, uint32_t phy_addr, uint32_t reg_addr, uint32_t val);


uint32_t phy_setup(void *priv);
uint32_t phy_reset(void *priv);        /* Perform the SW Reset of the transceiver */
uint32_t phy_check_id(void *priv, uint32_t id, uint32_t id_mask);    /* Check the product ID
*/
uint32_t phy_restart_aneg(void *priv);    /* Restart the autonegotiation process */
void phy_setup_forced(void *priv);        /* Force the link capabilities (speed and duplex)
*/
void phy_debug(void *priv);        /* dump the PHY registers */

uint32_t phy_get_link(void *priv);        /* Get the link status */
uint32_t phy_get_speed(void *priv);     /* Get the link speed */
uint32_t phy_get_mode(void *priv);         /* Get the link duplex mode */

void phy_set_link(void *priv, uint32_t link);
void phy_set_speed(void *priv, uint32_t speed);
void phy_set_mode(void *priv, uint32_t mode);
```

Below a snapshot of the dp83848_setup function. It will check the ID and it can restart the auto-negotiation process:

```
if (phy_reset(dpd->parent)) {
      return pdFALSE;
  }

  if (phy_check_id(dpd->parent, DP83848_ID, DP83848_MASK)) {
      return pdFALSE;
  }

  /* Auto-MDIX is supported too */
…

  val = phy_read(dpd->parent, dpd->addr, DP83848_RBR);
  if (dpd->mode == SPC5_PHY_MODE_RMII) {
      val |= DP83848_RBR_RMII_MODE;
  } else {
      val &= ~DP83848_RBR_RMII_MODE;
  }
  phy_write(dpd->parent, dpd->addr, DP83848_RBR, val);

  /* Disable all interrupts */
…
  /* loopback configuration is supported too*/
…

  if (phy_restart_aneg(dpd->parent)) {
      return pdFALSE;
  }
```

10 MAC low level driver APIs

These are the main APIs to setup the MAC core and enable the transmission and reception processes:

```
const struct dwmac_qos_core_ops core_ops = {
  .init = dwmac_qos_core_init,            /* Initialize the Core layer */
  .isr = dwmac_qos_core_isr,            /* Main interrupt service routine */
  .tx_enable = dwmac_qos_tx_enable,        /* Enable the transmission process */
  .rx_enable = dwmac_qos_rx_enable,        /* Enable the reception process */
  .set_mac_addr = dwmac_qos_set_mac_addr,    /* Set the MAC address */
  .get_mac_addr = dwmac_qos_get_mac_addr,
};
```

Refer to the SPC5Studio application demo for the complete function implementation while point to the Reference Manual (see Section Appendix B Reference documents) where all register description is available.

10.1 MAC initialization function

The core has to be configured before initializing the DMA and starting the RX/TX processes.
There are several settings that can be applied in the MAC Control register as shown below.

```
priv->reg->MAC_CONFIGURATION.B.ACS = 1;
priv->reg->MAC_CONFIGURATION.B.JD = 1;
/* MII mode */
priv->reg->MAC_CONFIGURATION.B.PS = 1;
/* HW Checksum */
priv->reg->MAC_CONFIGURATION.B.IPC = 1;

/* Mask the Interrupts */
priv->reg->MAC_INTERRUPT_ENABLE.B.RXSTSIE = 0;
priv->reg->MAC_INTERRUPT_ENABLE.B.TXSTSIE = 0;

/* Disable the LPI, Timestamp and PMT interrupts */
priv->reg->MAC_INTERRUPT_ENABLE.B.TSIE = 0;
priv->reg->MAC_INTERRUPT_ENABLE.B.LPIIE = 0;
priv->reg->MAC_INTERRUPT_ENABLE.B.PMTIE = 0;

/* Reset, init counters and disable the MMC interrupts */
priv->reg->MMC_CONTROL.B.CNTPRSTLVL = 1;
priv->reg->MMC_CONTROL.B.CNTPRST = 1;
priv->reg->MMC_CONTROL.B.RSTONRD = 0;
priv->reg->MMC_CONTROL.B.CNTRST = 1;
/* By default, disable the IRQ */
priv->reg->MMC_RX_INTERRUPT_MASK.R = 0xffffffff;
priv->reg->MMC_TX_INTERRUPT_MASK.R = 0xffffffff;
priv->reg->MMC_IPC_RX_INTERRUPT_MASK.R = 0xffffffff;


/* Enable queues */
priv->reg->MAC_RXQ_CTRL0.B.RXQ0EN = 0;
…
```

During the initialization phase it is necessary to provide the MAC addresses as shown below.
This Application Note doesn't cover perfect and hash filtering (e.g. to join to a Multicast network)
It is mandatory to store the data in the registers guaranteeing that, the low part of the address is programmed after the high one. This is to allow a synchronization inside the filter module in this peripheral.

```
et_mac_addr(dwmac_qos_t * priv, uint8_t addr[6])
...
 dataH = (addr[5] << 8) | addr[4];
 dataL = (addr[3] << 24) | (addr[2] << 16) | (addr[1] << 8) | addr[0];
 priv->reg->MAC_ADDRESS0_HIGH.B.ADDRHI = dataH;
 priv->reg->MAC_ADDRESS0_LOW.B.ADDRLO = dataL;
```

10.2 MAC Transaction Layer

The MAC Transaction Layer (MTL) provides the FIFO memory Interface to buffer and regulate the packets between the application system memory and the MAC.

The MTL layer has two data paths: Transmit path and Receive Path.

The following function initializes the MTL (MAC Transaction Layer) registers to:

- set the mode of operation to Store And Forward both for Tx and Rx;
- set the Threshold level of MTL Tx and Rx Queue (32 bit);
- enable the transmit Queue #0;
- set the Threshold level of MTL Rx Queue (64 bytes);
- set the size of allocated Transmit and Receive queues in blocks of 256 bytes.

```
priv->reg->MTL_TXQ0_OPERATION_MODE.B.TSF = 1;
priv->reg->MTL_TXQ0_OPERATION_MODE.B.TTC = 0;
priv->reg->MTL_TXQ0_OPERATION_MODE.B.TXQEN = 2;
fifosize = 128 << priv->reg->MAC_HW_FEATURE1.B.TXFIFOSIZE;
priv->reg->MTL_TXQ0_OPERATION_MODE.B.TQS = fifosize / 256 - 1;
priv->reg->MTL_RXQ0_OPERATION_MODE.B.RSF = 1;
priv->reg->MTL_RXQ0_OPERATION_MODE.B.RTC = 0;
fifosize = 128 << priv->reg->MAC_HW_FEATURE1.B.RXFIFOSIZE;
priv->reg->MTL_RXQ0_OPERATION_MODE.B.RQS = fifosize / 256 - 1;
```

10.3 DMA initialization function

The DMA in the Ethernet subsystem transfers data based on a linked list of descriptors.

The descriptors are created in the system memory by the network interface port layer.

The DMA has independent Transmit (Tx) and Receive (Rx) engines, and a CSR space.

The Tx engine transfers data from the system memory to the device port (MTL), whereas the Rx engine transfers data from the device port to the system memory.

The DMA transfers the data packets received by the MAC to the Rx Buffer in system memory and Tx data packets from the Tx Buffer in the system memory. The descriptors that reside in the system memory contain the pointers to these buffers.

Follow the hook functions for the DMA operations:

```
const struct dwmac_qos_dma_ops dma_ops = {
  .init = dwmac_qos_dma_init,
  .isr = dwmac_qos_dma_isr,
  .rx_watchdog = dwmac_qos_dma_rx_watchdog,
  .hw_features = dwmac_qos_hw_features,
  .set_tx_tail = dwmac_qos_set_tx_tail,
  .set_rx_tail = dwmac_qos_set_rx_tail,
  .start_tx = dwmac_qos_dma_start_tx,
  .start_rx = dwmac_qos_dma_start_rx,
};
```

The DMA initialization performs the following main steps:

- Perform a Software reset by setting the DMA_MODE.B.SWR field.
- Configure the DMA_SYSBUS_MODE.
- Initialize the MTL registers.
- Configure ring sizes inside the registers DMA_CH_TXDESC_RING_LENGTH and DMA_CH_RXDESC_RING_LENGTH.

- Set the Start of Receive / transmit Lists (DMA_CH_TXDESC_LIST_ADDRESS / DMA_CH_RXDESC_LIST_ADDRESS).
- Set the tail pointer registers, i.e.: DMA_CH_TXDESC_TAIL_POINTER and DMA_CH_RXDESC_TAIL_POINTER.

At the time of writing, only the DMA channel zero is managed and below the main steps to initialize a channel:

```
/* Enable the desired interrupt events */
priv->reg->DMA_CH[channel].DMA_CH_INTERRUPT_ENABLE.B.NIE = 1;
priv->reg->DMA_CH[channel].DMA_CH_INTERRUPT_ENABLE.B.AIE = 1;
priv->reg->DMA_CH[channel].DMA_CH_INTERRUPT_ENABLE.B.FBEE = 1;
priv->reg->DMA_CH[channel].DMA_CH_INTERRUPT_ENABLE.B.RIE = 1;
priv->reg->DMA_CH[channel].DMA_CH_INTERRUPT_ENABLE.B.TIE = 1;
priv->reg->DMA_CH[channel].DMA_CH_INTERRUPT_ENABLE.B.RBUE = 1;
priv->reg->DMA_CH[channel].DMA_CH_CONTROL.B.PBLX8 = 1;
priv->reg->DMA_CH[channel].DMA_CH_TX_CONTROL.B.TXPBL = 8;
priv->reg->DMA_CH[channel].DMA_CH_RX_CONTROL.B.RXPBL = 8;

/* Enable the operating on second frame. */
priv->reg->DMA_CH[channel].DMA_CH_TX_CONTROL.B.OSF = 1;

 /* Program the buffer len: e.g. 1536 (suitable for VLAN too). */
priv->reg->DMA_CH[channel].DMA_CH_RX_CONTROL.B.RBSZ = BUF_SIZE;

/* Program descriptor register to point to the tx and rx user rings */
priv->reg->DMA_CH[channel].DMA_CH_TXDESC_LIST_ADDRESS.R = (uint32_t) priv->dma_tx;
priv->reg->DMA_CH[channel].DMA_CH_RXDESC_LIST_ADDRESS.R = (uint32_t) priv->dma_rx;
```

10.4 MDIO access

The following API are called by the low layer to access PHY registers by using the Ethernet controller SMA interface.

- MDIO bus READ:

```
while (dwmac->reg->MAC_MDIO_ADDRESS.B.GB) {}

dwmac->reg->MAC_MDIO_DATA.B.RA = reg_addr;
dwmac->reg->MAC_MDIO_DATA.B.GD = 0;
dwmac->reg->MAC_MDIO_ADDRESS.B.PA = phy_addr;
dwmac->reg->MAC_MDIO_ADDRESS.B.RDA = reg_addr;
dwmac->reg->MAC_MDIO_ADDRESS.B.GOC = GMII_READ_OP;

dwmac->reg->MAC_MDIO_ADDRESS.B.GB = 1;

while (dwmac->reg->MAC_MDIO_ADDRESS.B.GB) {}

return (uint16_t) (dwmac->reg->MAC_MDIO_DATA.R & 0x0000FFFF);
```

- MDIO bus WRITE:

```
while (dwmac->reg->MAC_MDIO_ADDRESS.B.GB) {}

dwmac->reg->MAC_MDIO_DATA.B.RA = reg_addr;
dwmac->reg->MAC_MDIO_DATA.B.GD = data;
dwmac->reg->MAC_MDIO_ADDRESS.B.PA = phy_addr;
dwmac->reg->MAC_MDIO_ADDRESS.B.RDA = reg_addr;
dwmac->reg->MAC_MDIO_ADDRESS.B.GOC = GMII_WRITE_OP;

dwmac->reg->MAC_MDIO_ADDRESS.B.GB = 1;

while (dwmac->reg->MAC_MDIO_ADDRESS.B.GB) {}
```

The phy_addr is one the 32 possible addresses where the device is found while the reg_addr is the physical register to be accessed.

10.5 Descriptor debugging

When the Ethernet driver is initialized, the xNetworkInterfaceInitialise set up both transmits and receives rings, allocating the descriptors and all the buffers.

The descriptors are initialized to be used by the DMA engine at once; below the descriptor status from a real debug session.

The Descriptors 0 always points to the pre-allocated buffers, e.g.: des0 = 0x4002837C is the buffer managed by the descriptor of the first element in the transmit ring.

```
DMATxDesc = (
    (des0 = 0x4002837C, des1 = 0x0, des2 = 0x0, des3 = 0x0),
    (des0 = 0x4002897C, des1 = 0x0, des2 = 0x0, des3 = 0x0),
    (des0 = 0x40028F7C, des1 = 0x0, des2 = 0x0, des3 = 0x0),
    (des0 = 0x4002957C, des1 = 0x0, des2 = 0x0, des3 = 0x0),
    …
    (des0 = 0x40032B7C, des1 = 0x0, des2 = 0x0, des3 = 0x0),
    (des0 = 0x4003317C, des1 = 0x0, des2 = 0x0, des3 = 0x0),
    (des0 = 0x4003377C, des1 = 0x0, des2 = 0x0, des3 = 0x0),
    (des0 = 0x40033D7C, des1 = 0x0, des2 = 0x0, des3 = 0x0))
```

When the xNetworkInterfaceOutput() is invoked by the stack the descriptor will be prepared as shown below.

```
p = 0x40034D10 -> (
    des0 = 0x4002837C,
    des1 = 0x0,
    des2 = 0x8000002A,
    des3 = 0xB0030000)
```

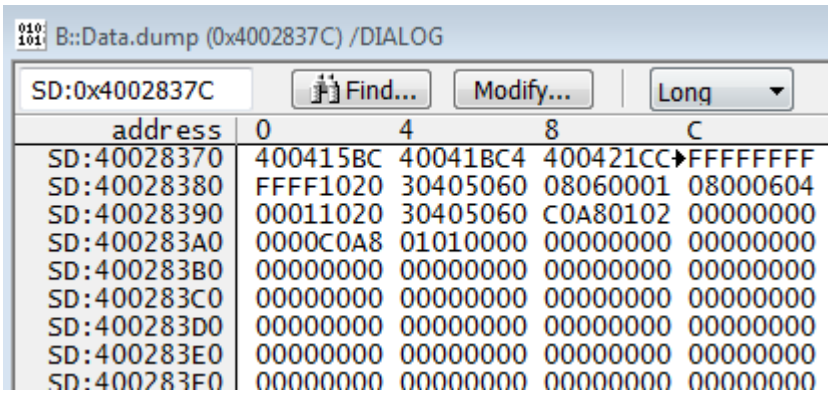
The des2 has to set the interrupt on completion bit and the size of the frame to be sent.

The des3 programs the FS, LS, CIC and OWN bits. So, the First and Last segment bits are set and the Hardware will perform the checksum (CIC: Checksum Insertion Control.). The OWN bit is set because the descriptor has to be managed by the DMA as soon as the TAIL pointer register is updated.

In this case, the xNetworkInterfaceOutput() is called to send a buffer of 42 bytes (0x2A) found at the address pointed by the des0 (0x4002837C).

This buffer, for example, contains a broadcast frame and the MAC address of the Ethernet device is 10:20:30:40:50:60.

Figure 13. Tx descriptor 0 dump



As soon as the packet is sent, the interrupt will notify the transmission status.

For example, below the status of the descriptor in write-back format: no error on the latest segment in this case.

```
(des0 = 0x4002837C, des1 = 0x0, des2 = 0x0, des3 = 0x10000000),
```

Concerning the reception ring, all the descriptors are configured by the xNetworkInterfaceInitialise to be handled by the DMA so, in the descriptor 3 the OWN bit is set and the Buffer1 is used.

```
DMARxDesc = (
    (des0 = 0x400366D4, des1 = 0x0, des2 = 0x0, des3 = 0xC1000000),
    (des0 = 0x40036CDC, des1 = 0x0, des2 = 0x0, des3 = 0xC1000000),
    (des0 = 0x400372E4, des1 = 0x0, des2 = 0x0, des3 = 0xC1000000),
    (des0 = 0x400378EC, des1 = 0x0, des2 = 0x0, des3 = 0xC1000000),
…
    (des0 = 0x40040FB4, des1 = 0x0, des2 = 0x0, des3 = 0xC1000000),
    (des0 = 0x400415BC, des1 = 0x0, des2 = 0x0, des3 = 0xC1000000),
    (des0 = 0x40041BC4, des1 = 0x0, des2 = 0x0, des3 = 0xC1000000),
    (des0 = 0x400421CC, des1 = 0x0, des2 = 0x0, des3 = 0xC1000000))
```

The prvNetworkInterfaceInput() is called by the ISR; it has to transfer a valid packet from the DMA buffer to the TCP stack.

Below there is the status descriptor managed by the DMA, in write back format:

```
DMARxDesc = (
(des0 = 0x0, des1 = 0x11, des2 = 0x0, des3 = 0x3401015A),
(des0 = 0x40036CDC, des1 = 0x0, des2 = 0x0, des3 = 0xC1000000),
(des0 = 0x400372E4, des1 = 0x0, des2 = 0x0, des3 = 0xC1000000),
```

From des1, the IPV4 header is detected on UDP packet.

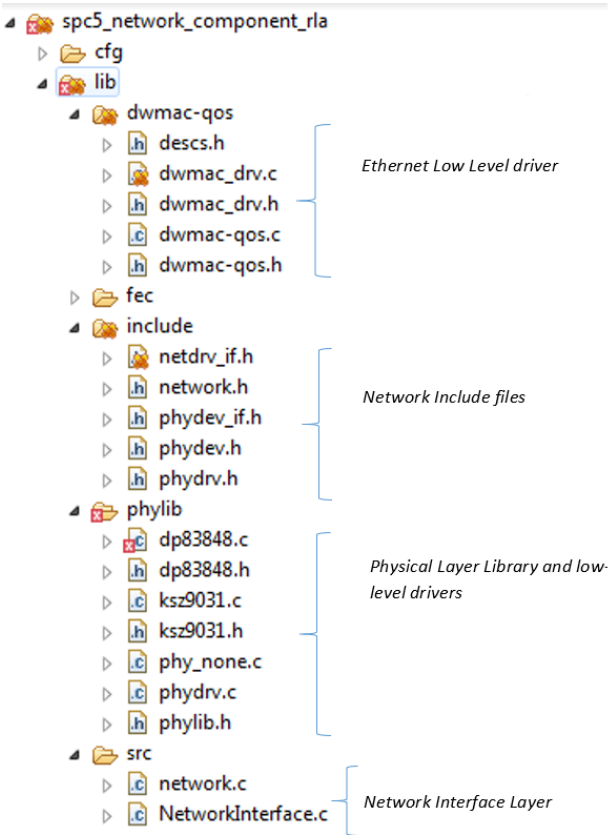
The des3 reports that is the latest segment, the OWN bit is 0 and there are no errors on a packet of 346 bytes.

The descriptors and buffers handled by the DMA engine, can be debugged by looking at the following registers: DMA_CH_CURRENT_APP_TXDESC, DMA_CH_CURRENT_APP_RXDESC, DMA_CH_CURRENT_APP_TXBUFFER, DMA_CH_CURRENT_APP_RXBUFFER

11 Networking component sources

From SPC5Studio, the source code will be available and the picture below shows the files organization inside the project:

Figure 14. Network compoent file structure



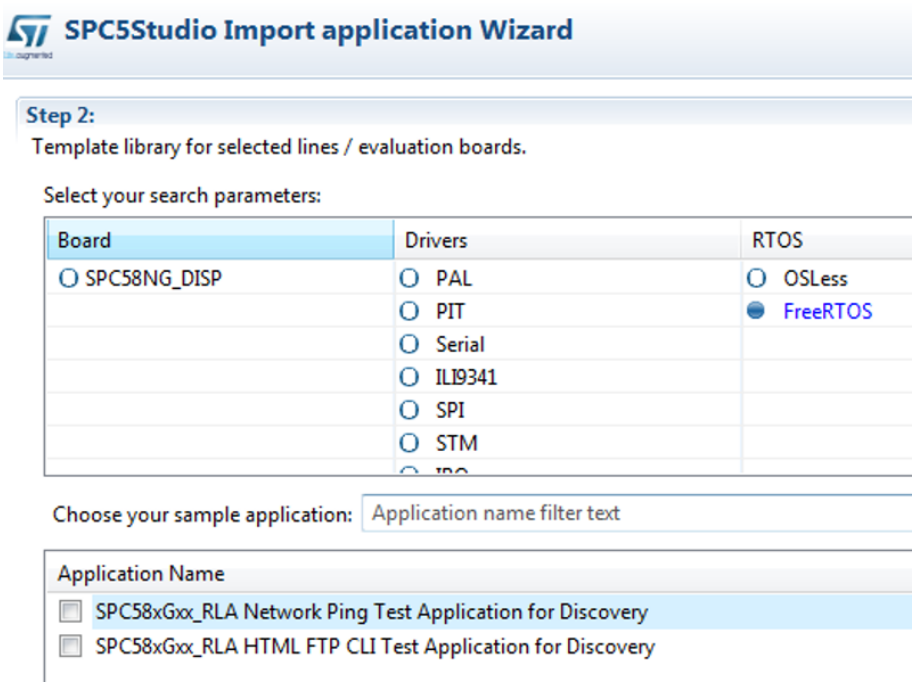
12 SPC5Studio Network applications demos

SPC5Studio offers different test applications for networking designed to start testing network protocols with a default environment where signals and clocks are setup and TCP and Ethernet stacks are configured.

Many parameters, as described in the previous chapters, can be tuned.

Some features can be disabled or enabled according to the user needs.

Figure 15. SPC5Studio wizard – SPC58xGxx OS Network demos



12.1 SPC5Studio Network ping application

This a basic application that has been designed to perform a first and basic test of the whole network stack. After negotiating the link, the Ethernet will be enabled; from a remote host machine it will be possible to ping the IP address assigned to the SPC58xx controller interface. So the ICMP protocol will be stimulated; reply to ICMP incoming packets is enabled by default.

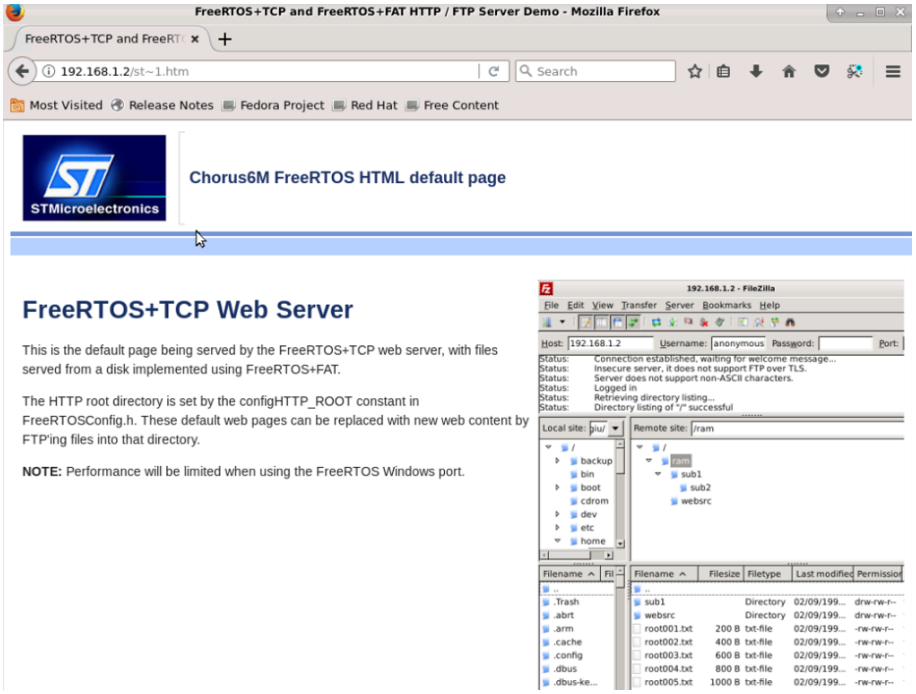
Figure 16. Ping from a remote Linux machine

```
ping -s 1000 192.168.1.2
PING 192.168.1.2 (192.168.1.2) 1000(1028) bytes of data.
1008 bytes from 192.168.1.2: icmp_seq=1 ttl=64 time=1.43 ms
1008 bytes from 192.168.1.2: icmp_seq=2 ttl=64 time=1.07 ms
^C
--- 192.168.1.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 1.071/1.253/1.436/0.185 ms
```

12.2 SPC5Studio HTML / FTP / CLI application

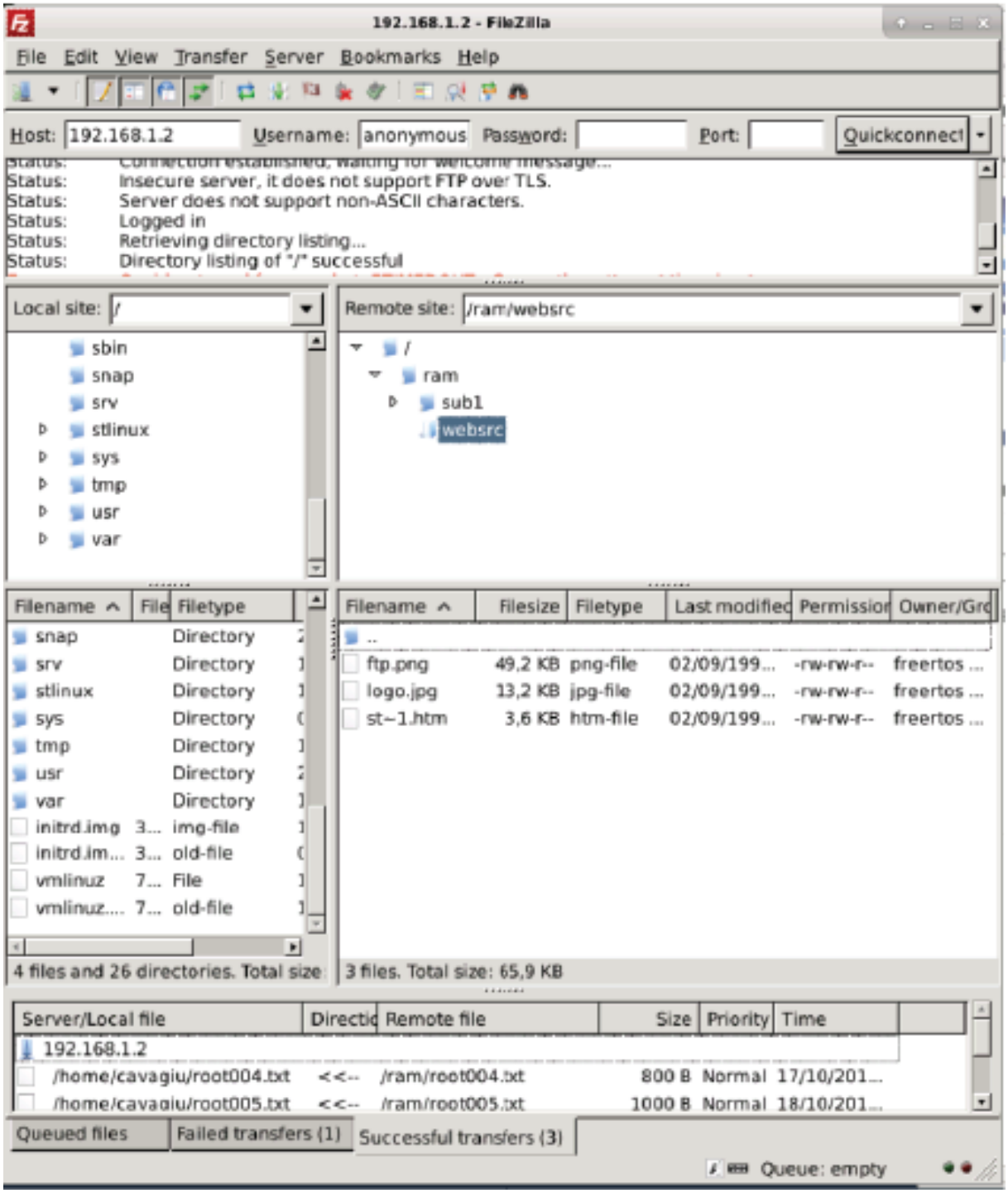
This application shows an advanced usage of the whole network stack, it will set up a web server so, as shown in the figure below, from a remote machine it will be possible to get the HTML web page from a browser:

Figure 17. Browsing the web page



From a remote host machine it will be also possible to open an FTP session and browse the RAM File System moving files from/to the board under testing.

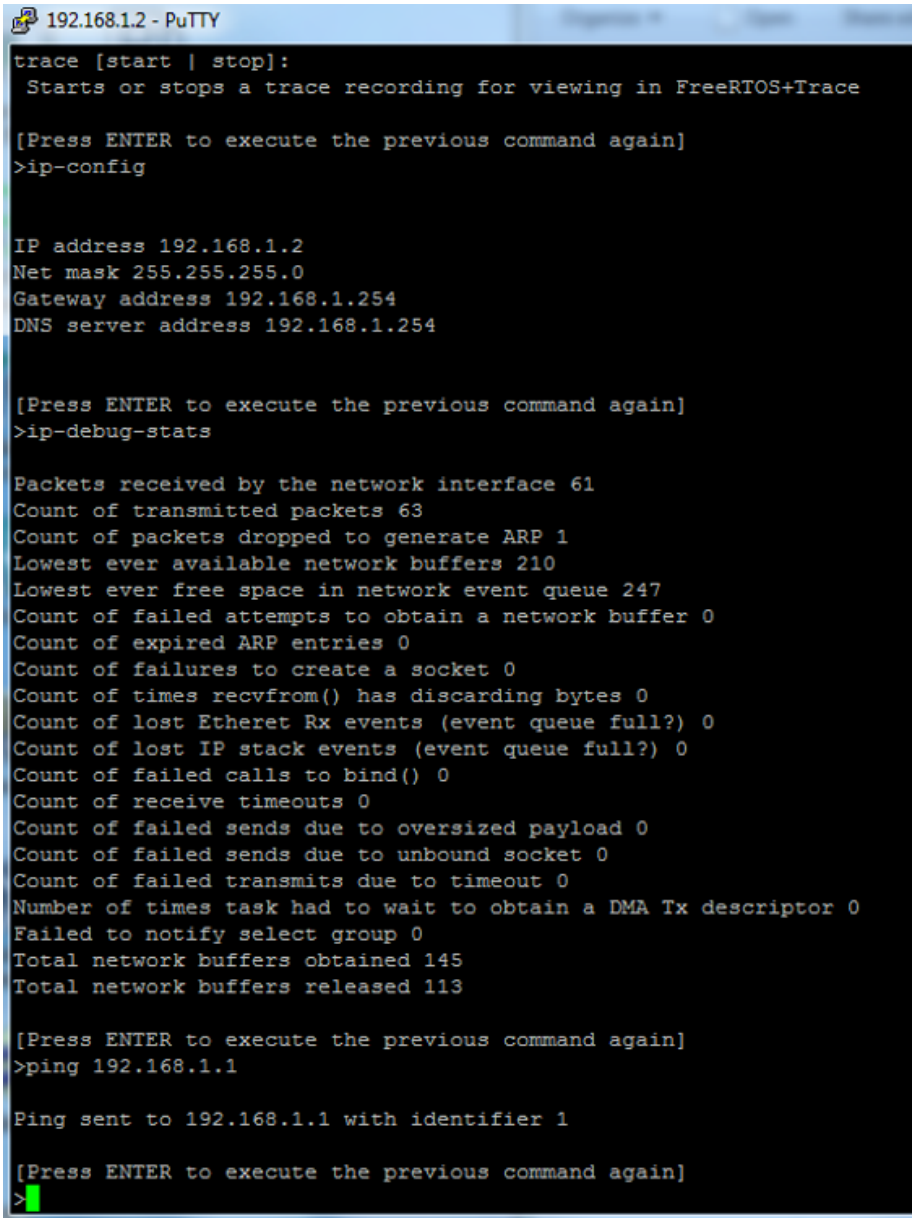
Figure 18. **FTP session**



FreeRTOS+CLI (Command Line Interface) provides a simple, small, and extensible method of enabling your FreeRTOS application to process command line input.

User can connect by using telnet command and a sub-set of commands are exposed to get statistics, to browse the RAMFS and to ping a remote host.

Figure 19. **Telnet session**



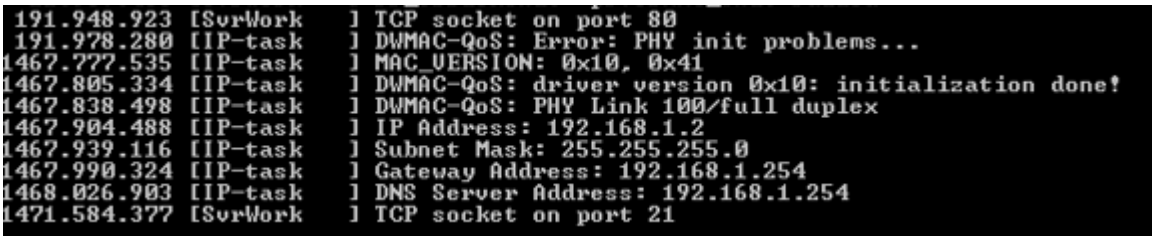
12.3 Log Messages via UDP

The UDP logging demonstrates how to send FreeRTOS_debug_printf() and FreeRTOS_printf() output to a UDP port. In the previous chapters it has been introduced its configuration through SPC5Studio, in this chapter it is shown how the FreeRTOS_debug_printf() and FreeRTOS_printf() are used to dump the application log messages. below is the set of defines that are configured:

```
/*
 * UDP Logging Configuration Settings
 */
#if( ipconfigHAS_PRINTF == 1 ) || ( ipconfigHAS_DEBUG_PRINTF == 1 )
#define configUDP_LOGGING_NEEDS_CR_LF                ( 1 )

#define configUDP_LOGGING_STRING_LENGTH                ( 200 )
#define configUDP_LOGGING_MAX_MESSAGES_IN_BUFFER        ( 20 )
#define    configUDP_LOGGING_TASK_PRIORITY                ( 2 )

/*
 * The UDP port to which the UDP logging facility sends messages.
 */
#define configUDP_LOGGING_PORT_REMOTE                1500
#define configUDP_LOGGING_PORT_LOCAL                1499

/*
 * Remote Host IP address
 */
#define configUDP_LOGGING_ADDR0                    192
#define configUDP_LOGGING_ADDR1                    168
#define configUDP_LOGGING_ADDR2                    1
#define configUDP_LOGGING_ADDR3                    1
#endif
```

On remote host, user can have a console by using netcat under Linux or udpterm_std under Windows (see output below):

Figure 20. UDP console output example



Note: tools mentioned above need to be configured to have the same UDP ports and IP server/client addresses.

12.4 Network benchmarks

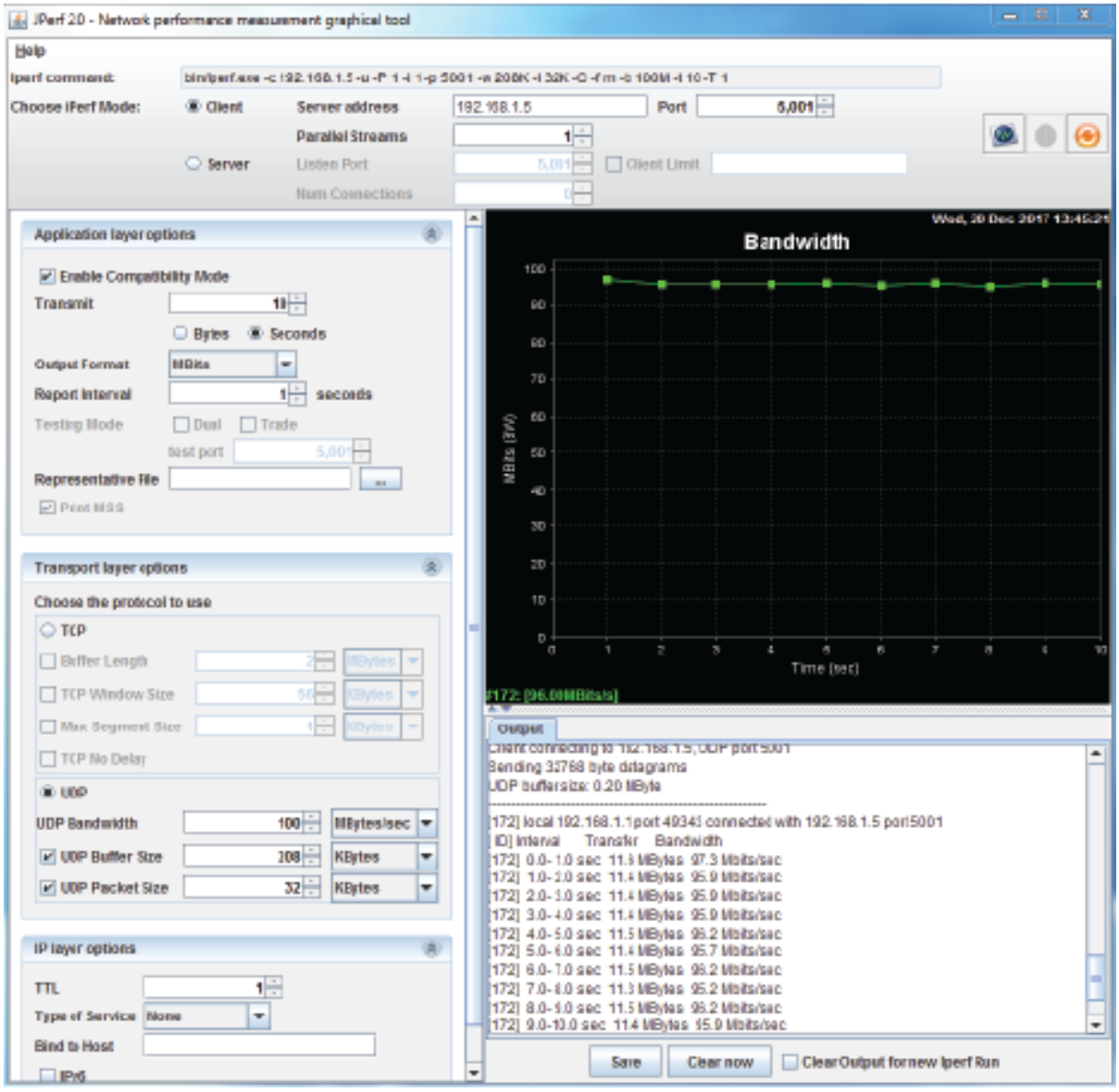
SPC5Studio embeds the well known ***iperf*** server in the benchmark component that can be optionally added and enabled by user.

This provides a TCP/UDP server running that allows the user to benchmark these protocols.

The jPerf is a graphical tool for active measurements of the maximum achievable bandwidth on IP networks. It supports TCP and UDP protocols, jPerf client runs on the host and not on the target microcontroller.

The figure below shows a jPerf session output and the related performances for UDP unidirectional reception where *~96Mbps* are measured.

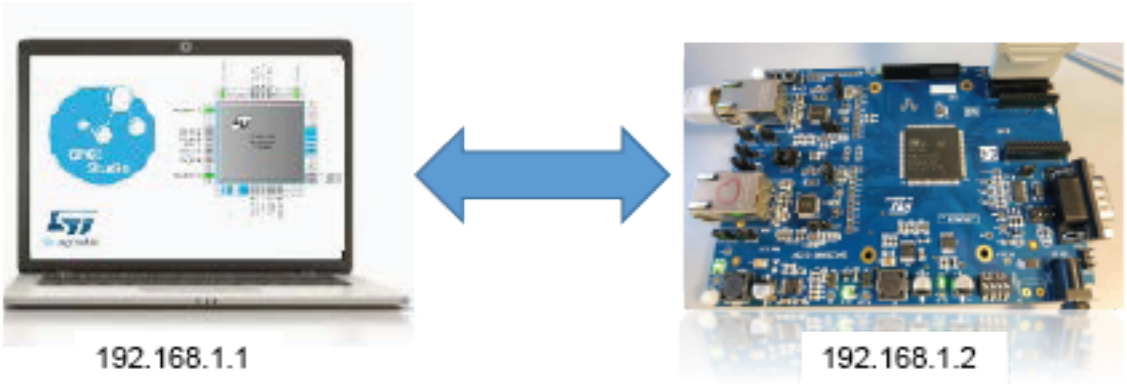
Figure 21. **jPERF output – UDP RX test throughput measurement**



13 Set up notes

The following figure shows the simple set up used for running all the applications.

Figure 22. Network set up



14 Conclusion

This application note can be considered a sort of Getting started for any Reader that wants to run a real network demo application on the SPC5x Automotive Microcontrollers.

The document reports the main parts to configure and have a full network stack working: TCP/IP, Ethernet and Physical Layer.

The example code could have some differences from what is included in the latest official tool releases because these are continuously improved in order to support new features, for example: Multiple Network Interface support.

In any case, the code showed in the document can be considered the base reference code for programming the Ethernet controller.

The application demos, based on FreeRTOS, also provide a real usage of the networking on SPC5x MCU showing many of the potentialities of this hardware (DMA engine, check-summing) and also some basic performance measurements are reported; that could be improved by applying further system (e.g. cache enable), network stack tuning and driver optimizations, e.g. zero copy.

Appendix A Acronyms and abbreviations

Table 1. Acronyms

Abbreviation	Complete name
MMI	Media independent interface
RMII	Reduced Media independent interface
GMII	GiGa Media independent interface
RGMII	Reduced GiGa Media independent interface
SGMII	Serial gigabit media independent interface
DMA	Direct Memory Address
QoS	Quality of Service
PCS	Physical Coding SubLayer
MAC	Medium Access Control
MTL	MAC Transaction layer
ISR	Interrupt Service Routine
EEE	Energy Efficient Ethernet
LPI	Low-Power Idle
WoL	WakeUp on Lan
MMC	MAC Management Counters
SMA	Station Management Agent
L<x>	Network Layer, e.g. L3: IPv4, L4: TCP

Appendix B Reference documents

- Reference manual: RM0407
- Reference manual: RM0421
- Technical note: TN1305
- DP83848C Single Port 10/100 Mb/s Ethernet Physical Layer Transceiver

Revision history

Table 2. **Document revision history**

Date	Version	Changes
26-Jun-2020	1	Initial release.

Contents

List of figures