life.augmented

AN5697

Application note

SR6 P7/G7 line - Ethernet raw interface

Introduction

SR6Px automotive micro-controllers features a quality-of-service 10/100/1000 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 explains how to start with and use the raw application programming interface on top of Ethernet device driver provided in SR6Px software development kit. This document describes the functions provided by this simple API and gives simple example to initialize the device, then read, write data using this API.

1 Networking Raw support

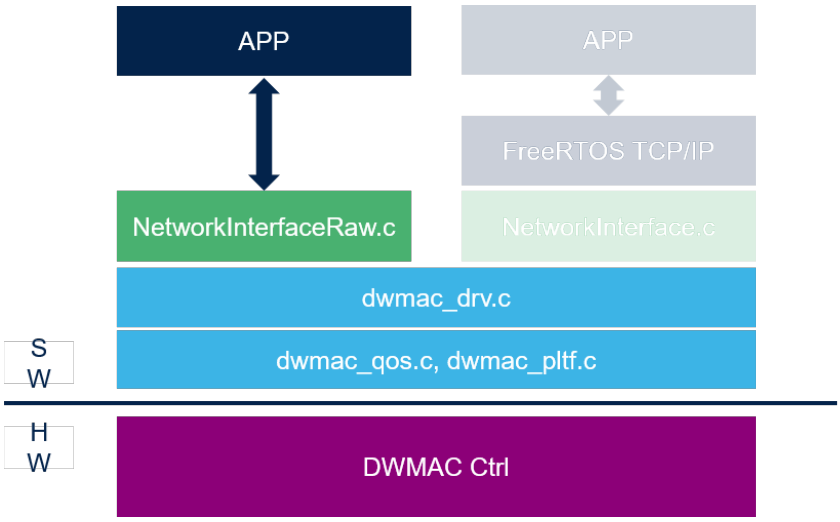
There are two network supports available on top of Ethernet device driver provided in SDK development framework:

- The standard interface that connects a FreeRTOS TCP/IP stack to Ethernet device driver (aka: NetworkInterface.c).
- The raw interface that allow any application to directly read and write raw data from/to Ethernet device driver. This API may be used to perform very basic simple network use-cases that do not require a network stack between the application and the Ethernet device driver.

Whenever an application wants to rely on the raw network API, considering that there is no network stack between the application and the Ethernet device driver, it is important to keep in mind that it's up to the application to handle the encoding and decoding of Ethernet packets. In other words, when relying on the raw support:

- There is no software support for internet and transport layers that make up one typical network stack. So this is under the responsibility of the application to implement these layers or not.
- As a matter of fact, sophisticated networking features like for instance relying on Ethernet Sockets is obviously not available in this case.

Figure 1. Software stack overview



As example, the raw API could be used to implement a basic test that would measure the raw network throughput of Ethernet controller either using its internal MAC loop-back, or connecting it to another network device in a point to point way.

2 Raw API public functions

Here below the list of RAW APIs available as well in *Ethernet/include/network_raw.h*, file which shall be included in application source file.

Table 1. API prototypes

Public Prototypes	Purpose
`unsigned int rawNetworkInterfaceInitialise(uint32_t interface, const uint8_t *mac, bool mac_loopback)`	Network device initialization
`unsigned int rawNetworkInterfaceRegisterRx(uint32_t interface, void (*rx_cb)(void **buf, uint32_t len))`	Input frame registering
`unsigned int rawNetworkInterfaceOutput(uint32_t interface, uint32_t ulTransmitSize, uint8_t *pucEthernetBuffer)`	Frame transmission

2.1 rawNetworkInterfaceInitialize

This function initializes an Ethernet device given an interface id, a mac address and a boolean to set whether or not, internal loop-back shall be enabled.

- `uint32_t interface`: id of Ethernet device to be initialized. On SR6P6/P7 that embed two Ethernet controllers, this parameter can be set to one of those definitions from *Ethernet/include/network_cfg.h* ressource file:

```
#define SSDK_ETH0_INSTANCE 0U
#define SSDK_ETH1_INSTANCE 1U
```

- `const uint8_t *mac`: pointer to array of uint8_t that store the mac address that device shall use.
- `bool mac_loopback`: if set to true, internal mac loopback inside the device will be enabled.

This function returns 1U in case device was successfully initialized and 0U if not.

Pay attention that following switches in *Ethernet/include/network_cfg.h* must be properly configured:

```
#define SSDK_USE_ETH0 TRUE/FALSE
#define SSDK_USE_ETH1 TRUE/FALSE
```

As example, to initialize Ethernet1 using internal mac loopback, ensure that `SSDK_USE_ETH0` is set to `TRUE` in *Ethernet/include/network_cfg.h* and then add this code to the application main function:

```
#include <network_raw.h>
int main(void) {
uint8_t mac[6] = {0x11, 0x22, 0x33, 0x44, 0x55, 0x66};
uint32_t interface = SSDK_ETH0_INSTANCE;
…
ret = rawNetworkInterfaceInitialise(interface, mac, true);
if (ret == 0U) {
return 1;
}
…
return 0;
}
```

2.2 rawNetworkInterfaceRegisterRx

This function registers a rx consumer call-back, given an interface id and a pointer to own application function (invoked by the Ethernet driver to consume the incoming frames).

- `uint32_t interface` : id of Ethernet device (either `SSDK_ETH0_INSTANCE` or `SSDK_ETH1_INSTANCE`)
- `void (*rx_cb)(void **buf, uint32_t len)` : pointer to rx consumer call-back function. Whenever a incoming frame is received by the Ethernet device with given id, rx consumer call-back will be called with the following arguments filled:
  - `void **buf`: pointer to rx buffer containing the incoming frame that has been received.
  - `uint32_t len`: frame length in bytes.

This function returns 1U in case rx consumer call-back was successfully registered and 0U if not.

Pay attention to the following things:

- Ethernet device must be initialized before calling this function.
- Rx consumer call-back is called by Ethernet driver interrupt service routine. Special care shall be taken when implementing things in rx consumer call-back to avoid masking interrupts for too much time. For instance time consuming tasks like frame decoding shall be handled later on outside interrupt context.
- If `DWMAC_RX_ZERO_COPY` and `DWMAC_MANAGE_RX` switches are enabled when building Ethernet driver, then it's up to the application to free rx buffer once it is no more used, using proper memory allocator API. See *Ethernet/dwmac-qos/dwmac_drv_helper.h* for details.
- If `Ethernet_RX_RAW_BUFFER_CONSUMER` is enabled, then a default rx consumer external call-back whose name is `Ethernet_consume_rx_buff()` will be automatically registered at the time `rawNetworkInterfaceInitialise()` is called. In such a case, it's up to the application to implement this function. Note that in case `rawNetworkInterfaceRegisterRx()` would be called later on (which is not advised), then rx call-back pointer pointing to `Ethernet_consumer_rx_buff()` call-back will be overwritten with the new call-back given by `rawNetworkInterfaceRegisterRx()`.

The sample source code below shows how to register a rx consumer call-back thanks to `rawNetworkInterfaceRegisterRx()` function:

```
#include <network_raw.h>
static void raw_test1_rx_cb(__attribute__((unused)) void **buf,
__attribute__((unused)) uint32_t len)
{
/* do something with buf and len … */
}
int main(void)
{
    uint8_t mac[6] = {0x11, 0x22, 0x33, 0x44, 0x55, 0x66};
    uint32_t interface = SSDK_ETH0_INSTANCE;
    ret = rawNetworkInterfaceInitialise(interface, mac, true);
    if (ret == 0U) {
        printf("%s: failed to init eth%d\r\n", __func__, interface);
        return 1;
    }
    ret = rawNetworkInterfaceRegisterRx(interface, raw_test1_rx_cb);
    if (ret == 0U) {
        printf("%s: failed to register rx callback\r\n", __func__);
    return 1;
    }
    …
    return 0;
}
```

2.3 rawNetworkInterfaceOutput

This function transmits a new frame to an Ethernet device, given an interface id, a frame size and a pointer to the buffer to be transmitted:

- `uint32_t interface` : id of Ethernet device (either `SSDK_ETH0_INSTANCE` or `SSDK_ETH1_INSTANCE`).
- `uint32_t ulTransmitSize`: size of buffer to be transmitted in bytes.
- `uint8_t *pucEthernetBuffer`: pointer to buffer to be transmitted.

This function returns 1U in case frame was successfully added in driver's tx ring. In case it returns 0U, this means there is currently no room in driver's tx ring. In such a case, application shall retry to send that frame.

Pay attention to the following things:

- ulTransmitSize shall not be greater than `BUF_SIZE` constants that is defined in Ethernet/include/network_cfg.h and that is set `1536UL` by default.
- In case `DWMAC_TX_ZERO_COPY` and `DWMAC_MANAGE_TX` are enabled, it's up to the application to allocate memory pointed by `pucEthernetBuffer` that will be passed to rawNetworkInterfaceOutput(). Ethernet driver will attempt to release memory pointed by `pucEthernetBuffer` on tx completion. See `dwmac_os_release_tx_zerocopy()` in *Ethernet/dwmac-qos/dwmac_drv_helper.h* for details.
- In case `DWMAC_TX_ZERO_COPY` but `DWMAC_MANAGE_TX` is disabled, Ethernet driver won't attempt to release memory pointed by pucEthernetBuffer on tx completion. This configuration might be used in case tx buffer points to a static array of bytes (so not dynamically allocated), and hence that must not be freed. But in such a case, some care must be taken at the application side when re-using the buffer. Since data inside buffer might not yet be transmitted at the time it is updated by the application.

The sample source code below shows how to output a frame over Ethernet0 thanks to `rawNetworkInterfaceOutput()` function:

```
#include <network_raw.h>
#include <string.h>
struct raw_udp_packet {
uint8_t d_addr[6];
uint8_t s_addr[6];
uint8_t type[2];
uint8_t ip_vers[1];
uint8_t ucDifferentiatedServicesCode[1];
uint8_t usLength[2];
uint8_t ioff[4];
uint8_t ucTimeToLive[1];
uint8_t ucProtocol[1];
uint8_t csum[2];
uint8_t s_ipaddr[4];
uint8_t d_ipaddr[4];
uint8_t udp_s_port[2];
uint8_t udp_d_port[2];
uint8_t Length[2];
uint8_t data[1500];
};
#define USER_MAC_ADDR {0x11, 0x22, 0x33, 0x44, 0x55, 0x66}
static struct raw_udp_packet udp_pkt = {
    USER_MAC_ADDR , /* Ethernet dest MAC address */
    USER_MAC_ADDR , /* Ethernet source MAC address */
    {0x08, 0x00}, /* Ethernet frame type */
    {0x45}, /* ucVersionHeaderLength */
    {0x00}, /* ucDifferentiatedServicesCode */
    {0x05, 0xda}, /* usLength: 1498 */
```

```
    {0x00, 0x00, /* usIdentification */
    0x00, 0x00}, /* usFragmentOffset */
    {0x80}, /* ucTimeToLive = 128 */
    {0x11}, /* ucProtocol = UDP */
    {0x00, 0x00}, /* usHeaderChecksum */
    {0xC0, 0xA8, 0x01, 0x04}, /* 192.168.1.4 Target IP addr */
    {0xC0, 0xA8, 0x01, 0x64}, /* Host address : e.g. : 192.168.1.100 */
    {0x13, 0x89}, /* UDP source port 5001 - iperf */
    {0xCB, 0x36}, /* UDP dest port 52022 */
    {0x05, 0xC6}, /* length 1478 */
    { 0xAA } /* data */
};
int main(void) {
    uint8_t mac[6] = USER_MAC_ADDR ;
    uint32_t interface = SSDK_ETH0_INSTANCE;
    struct raw_udp_packet *rup = &udp_pkt;
    ret = rawNetworkInterfaceInitialise(interface, mac, true);
    if (ret == 0U) {
        printf("%s: failed to init eth%d\r\n", __func__, interface);
        return 1;
    }
    memset(rup->data, 0xAAU, sizeof(rup->data));
    while (rawNetworkInterfaceOutput(interface, 1512, (uint8_t *)rup) == 0U) {
        printf("%s: failed to send frame over eth%d, retrying…\r\n", __func__,
        interface);
    }
    …
    return 0;
}
```

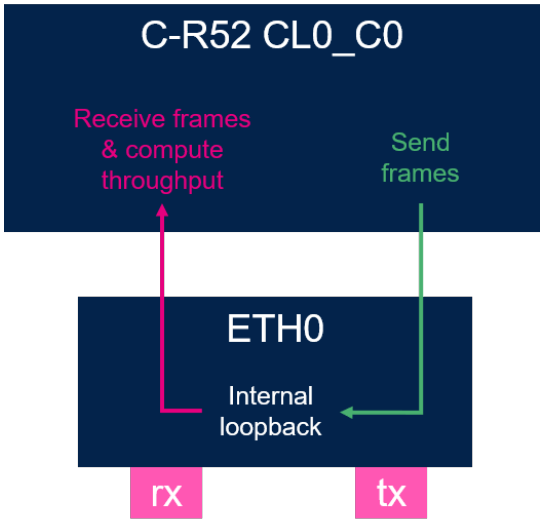
3 Examples

Ethernet Module available in Stellar Development Kit has a “*test”* directory. “*raw_ethernet_test.c*” in this directory contains test-cases (given as example) that make use of the Ethernet RAW Network Interface. Two simple test-cases are covered by this document, these have been designed to analyze the best throughput performances without FreeRTOS and TCP stack on different speed modes and frame sizes.

3.1 Single Ethernet in loopback mode

This test uses a single Ethernet device and it uses the device internal mac loop-back, the application sends frames with a specific size in loops. Time to send and receive an expected number of frames is measured to compute a throughput in Mbits/s. TEST_ETHERNET_RAW_ENABLE and CONFIG_RAW_ETH_TEST1 switches must be set to build corresponding source code for this test.

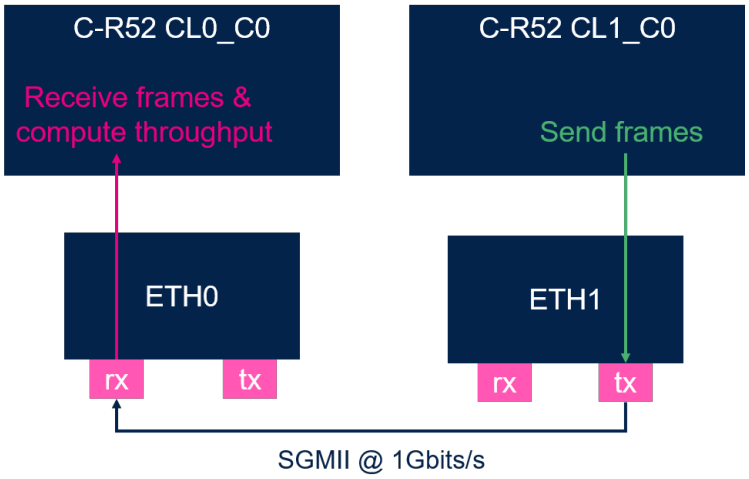
Figure 2. ETH0 internal loopback example



3.2 MAC2MAC testing

This test uses the two Ethernet devices embedded in the MCU, each device being driven by one C-R52 core. Note that this test can only be run on SR6Px SoC variants that embed two Ethernet devices. Ethernet0 and Ethernet1 must be bounded together. One core sends frames with a specific size in loops over Ethernet1 while the other core receives incoming frames from Ethernet0. Time to receive an expected number of frames is measured to compute a throughput in Mbits/s. TEST_ETHERNET_RAW_ENABLE & CONFIG_RAW_ETH_TEST2 switches must be set to build corresponding source code for this test. Moreover, CONFIG_RAW_ETH_TEST2_PERF_SERVER must be enabled in firmware of C-R52 that receive frame from Ethernet0 while CONFIG_RAW_ETH_TEST2_PERF_CLIENT must be enabled in firmware of C-R52 that sends frames over Ethernet1. Finally, SSDK_USE_ETH0 shall be to TRUE and SSDK_USE_ETH1 to FALSE in firmware of C-R52 that drives Ethernet0 while SSDK_USE_ETH0 shall be to FALSE and SSDK_USE_ETH1to TRUE in firmware of C-R52 core that drives Ethernet1.

Figure 3. **Testing two Ethernet controllers in MAC2MAC mode**



4 Conclusions

The full networking stack is one of the most complex and useful part of the SDK application that can be run on this micro-controller family. This is based on Operating System like FreeRTOS and related TCP/IP stack. This aims to demonstrate a real usage of the Ethernet controller. It also helps to analyze TCP/UDP performances, get statistics and implements several types of applications based on different protocols (HTML, FTP etc.) and implements more complex application like automotive gateways.

The network device drivers have been designed to be used without any top level stacks and, actually the raw support APIs offer a way to test the pure Ethernet layer in a stand-alone application so without any external dependency. Or the drivers can be used with own TCP software stack.

This support has been developed to offer a way to verify and implement ad-hoc tests on top of the only Ethernet stack, allowing Developer to study and analyze the Ethernet controller internals on this MCU but also to implement own robustness and performance stand-alone applications as demonstrated in the document.

Appendix A Acronyms

Table 2. List of acronysms

Acronym	Meaning
API	Application Programming Interface
SDK	Software development kit
MAC	Medium Access Control
DMA	Direct Memory Access

Appendix B Reference documents

Table 3. **Reference document**

Document name	Document number
SR6P7x, SR6G7x 32-bit ARM® Cortex®-R52 architecture microcontroller for automotive ASIL-D	RM0482

Revision history

Table 4. **Document revision history**

Date	Revision	Changes
21-Sep-2021	1	Initial release.

Contents