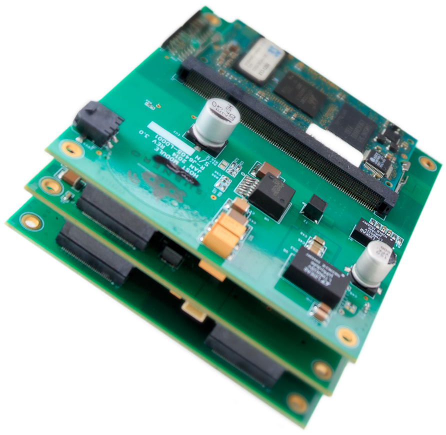
Subnero Underwater Modem Co-processor User Manual



M25M custom series

v1.0

Read this manual along with the modem user manual.

For the latest version of the user manuals, please visit:

`https://www.subnero.com/support/wnc/manuals`

November 2, 2018

CHAPTER 1

Co-processor specifications

The co-processor is a NVIDIA Jetson TX1 module.

Specifications	**Jetson TX1**
CPU	Quad ARM A57/2 MB L2
Memory	4 GB
GPU	NVIDIA Maxwell, 256 CUDA cores
OS	NVIDIA Linux-for-Tegra R28.2 (Linux Kernel 4.4.38)

The operating system installed on the co-processor is a standard NVIDIA Linux-for-Tegra install with a couple of extra configurations as listed below:

1. A daemon has been added to ensure that the connection between the co-processor and the modem stays alive.
2. Examples in `C`, `Python` and `Groovy/Java` on how to connect to the modem have been installed in `/home/ubuntu/Examples`
3. A new `host` called `unet-modem` has been added to point to the IP address of the modem.

CHAPTER 2

Accessing co-processor and modem

The custom modem built encloses a modem along with a co-processor as shown in Fig. 2.1. The co-processor is connected to the modem using a network interface. The external interface exposes the ethernet and power connectors to the modem.

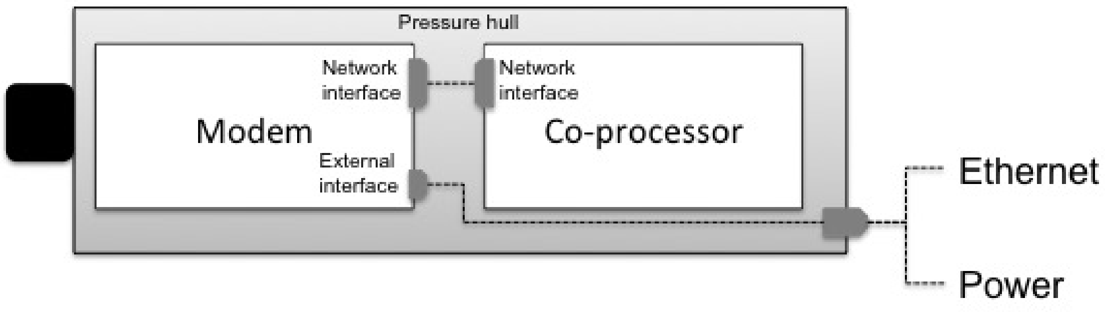


Figure 2.1: Block diagram of the custom modem with co-processor.

2.1 Accessing co-processor

The co-processor is accessible over SSH over port 2222 on the same IP address as the modem.

For example, if the modem is assigned the local IP address 192.168.0.1, then the co-processor can be accessed using

```
ssh -p 2222 192.168.0.1
```

The `username:password` for logging in to the co-processor is `ubuntu:ubuntu`.

Note: Currently, only the SSH port (22) of the co-processor is forwarded through the modem.

2.2 Accessing modem from co-processor

The modem can be accessed from the co-processor using any of the standard Unet APIs.

Examples in `C`, `Python` and `Groovy/Java` on how to connect to the modem have been installed in `/home/ubuntu/Examples`.

The modem is accessible from the co-processor over TCP on the IP address `192.168.8.1`. This has been added to the `/etc/hosts` file under the alias `unet-modem`. This allows the use of `unet-modem` in all examples and API calls.

CHAPTER 3

Developing applications on co-processor

The co-processor can be accessed by the user as explained in the previous chapter. Once logged in to the co-processor, the user can develop their applications or algorithms to test/run on the modem.

3.1 UnetStack overview

The modem is installed with `UnetStack` software. The `UnetStack` architecture defines a set of software agents that work together to provide a complete underwater networking solution. Agents play the role that layers play in traditional network stacks. However, as the agents are not organized in any enforced hierarchy, they are free to interact in any way suitable to meet application needs. This promotes low-overhead protocols and cross-layer information sharing. The stack runs on a Java virtual machine and `fjage` source agent framework. A detailed documentation of `UnetStack` is at the following link `https://www.unetstack.net/docs.html`.

3.2 Gateway functionality to interact with modem

`fjage` provides a `Gateway` class. This class provides the user to communicate with the agents running in `UnetStack` on the modem. This class is utilized and the APIs are developed for the user to build their application upon. The APIs to interact with the modem from any computer (in this case the co-processor) are available in `Groovy`, `C` and `Python`.

3.2.1 Opening a connection to modem

In order to open a connection to the modem using `Gateway` class, the modem's IP address and the port number are needed. The `UnetStack` runs on port number 1100 by default. An example in `Groovy` to open this connection is as shown below:

```
Gateway modem = new Gateway(ip_address, 1100)
```

The instance `modem` created can then be used to access all the methods provided by the `Gateway` class to interact with the modem. The `Gateway` class methods are documented at `http://org-arl.github.io/fjage/javadoc/`.

Note that the same interfaces are also available in `Python` and `C`. For example, to open a connection to the modem using `Python` is as shown below:

```
modem = UnetGateway(ip_address, 1100)
```

and in `C` is as shown below:

```
modem_t modem = modem_open_eth(ip_address, 1100);
```

3.3 Examples of basic operations

Once a connection is open to the modem, the user can write code to develop their own applications. Sample code in `C`, `Python` and `Groovy/Java` on how to connect to the modem and perform basic operations have been copied in `/home/ubuntu/Examples` folder for reference. Few basic operations are listed below and explained if the code is developed in `Groovy`:

1. *Transmit a frame containing data using FH-BFSK modulation (default CONTROL) scheme.*

   ```
   // Look for agents providing physical service
   def phy = modem.agentForService Services.PHYSICAL

   // Transmit a CONTROL packet
   ```

```
def msg = new TxFrameReq(type: Physical.CONTROL, data: 'hello' as byte[])
msg.recipient = phy
modem.send(msg)
```

The first step in transmitting a packet is to figure out which agent running in `UnetStack` provides a `Physical` service. The piece of code

```
// Look for agents providing physical service
def phy = modem.agentForService Services.PHYSICAL
```

looks for such agent and returns the `AgentID`. Now, the second step is to create a message supported by this agent to transmit data. The `TxFrameReq` is one such message which supports transmission of data using either CONTROL or DATA modulation scheme. In order to transmit, first the message is created

```
// Create a message containing data
def msg = new TxFrameReq(type: Physical.CONTROL, data: 'hello' as byte[])
```

and then the recipient of the message is set to the `AgentID` which provides the `Physical` service. We set it as recipient of the message, since that is the agent which can transmit the packet.

```
// set the appropriate recipient of the message
msg.recipient = phy
```

Finally, the message is sent to the `UnetStack` running on the modem which transmits the packet:

```
// send the message
modem.send(msg)
```

2. *Transmit a frame containing data using OFDM modulation (default DATA) scheme.*

```
// Look for agents providing physical service
def phy = modem.agentForService Services.PHYSICAL
def msg = new TxFrameReq(type: Physical.DATA, data: 'hello' as byte[])
msg.recipient = phy
modem.send(msg)
```

This code is similar to the one explained above for the CONTROL scheme, except that the message is created with the type DATA instead of CONTROL.

3. *Transmit a baseband signal.*

   Sometimes a user might want to create their own signal with custom modulation scheme and transmit using the modem. This is possible with the modem running `UnetStack`. The user can create a baseband signal as an array with alternate real and imaginary values and the standard Unet API can be used to transmit this baseband signal. An example code for performing such an operation is as shown below:

```
// Look for agents providing baseband service
def bb = modem.agentForService Services.BASEBAND

// Generate a baseband signal
float freq = 5000
float duration = 1000e-3
int fd = 24000
int fc = 24000
int n = duration*fd
def signal = []
(0..n-1).each { t ->
    double a = 2*Math.PI*(freq-fc)*t/fd
    signal << (int)(Math.cos(a))
    signal << (int)(Math.sin(a))
}

// Transmit a baseband signal
def msg = new TxBasebandSignalReq(preamble: 3, signal: signal)
msg.recipient = bb
modem.send(msg)
```

The first step again in transmitting a signal is to look for agent in `UnetStack` which provides `Baseband` service. The following piece of code

```
// Look for agents providing baseband service
def bb = modem.agentForService Services.BASEBAND
```

looks for such an agent and returns the `AgentID`.

The next step is to generate a baseband signal. In order to generate the baseband signal, the only thing to keep in mind for the user is to use a baseband sampling rate $f_d = 24000$ Hz. The carrier frequency of the modem by default is set at $f_c = 24000$ Hz. A sample code to generate a baseband signal is as shown below:

```
// Generate a baseband signal
float freq = 5000
float duration = 1000e-3
int fd = 24000
int fc = 24000
int n = duration*fd
def signal = []
(0..n-1).each { t ->
    double a = 2*Math.PI*(freq-fc)*t/fd
    signal << (int)(Math.cos(a))
    signal << (int)(Math.sin(a))
}
```

Note that the real and imaginary values of each sample are placed alternately in the `signal` array.

Next step is to create the message with the signal. This can be performed as shown below:

```
// Create a message containing the signal
def msg = new TxBasebandSignalReq(preamble: 3, signal: signal)
```

This example is using a specific preamble already available in the modem. In case, the user wants to include their own preamble they can do so and include it in the baseband signal generated.

Finally, the appropriate recipient for the message is set and the message is sent to the `UnetStack` which instructs the modem to transmit this signal as shown below:

```
// set the recipient and send the message
msg.recipient = bb
modem.send(msg)
```

4. *Record a baseband signal.*

   Finally, it is also possible to record a baseband signal. Upon a request to record the baseband signal, the modem records a passband signal and converts it to the appropriate baseband signal and returns it to the user. This operation is as shown below:

```
/ Record a baseband signal
def msg = new RecordBasebandSignalReq(recLen: 24000)
msg.recipient = bb
modem.send(msg)
```

   The above code should be easy to understand now. A message is created with recording length of 24000 baseband samples, i.e., this message requests the modem to record 24000 baseband samples from the current time. Note that a recording time in the past can also be set as a parameter in this API. For details visit the UnetStack documentation.

   Once the modem is instructed to record the baseband signal and the modem agrees to do so, a `RxBasebandSignalNtf` message will be sent out by the agent providing `Baseband` service notifying that the recording was performed successfully and it also contains the recorded signal. Therefore, a user can look for the reception of this message and extract the received signal from it as shown below:

```
// Receive the notification when the signal is recorded
def rxntf = modem.receive(RxBasebandSignalNtf, 5000)
if (rxntf != null) {
    // Extract the recorded signal
    def rec_signal = rxntf.signal
    println 'Recorded signal successfully!'
}
```

Support

- For support on UnetStack and related API usage, user can post questions on StackOverflow `https://stackoverflow.com/questions/tagged/unetstack`.
- For any technical queries on the modem operations, contact us at: `support@subnero.com`