How To Develop Driver, Part 3

In the third part of the lesson about Driver development, we will explain what is Packet Processor and how it handles different types and categories of packets.

Packet Processor is a module of Framework 4, whose task is to ensure the logical layer of communication with the device. It manages the entire lifecycle of packets created in the Device Logic. That means, sending packets and subsequent pairing of the received packets, as well as delivering them for further processing to the correct part of the Device Logic code developed by the Driver developer.

Also, it notifies the Device Logic about changes in the status of the communication, such as in the situations when an external system doesn't respond.

The last important task is to notify the C4 Server directly about the changes in the status of the communication with the device.

The internal architecture of Packet Processor consists of two main parts.

The first one is the so-called Packet Queue. Its task is to manage all packets that need to be sent to the device.

The second part is Processor, which takes care of sending data to the Link module on one side, and receiving packets from the Link module on the other side. The packets are then processed based on their type.

After creating the Link, we proceed with creating and initializing the Packet Processor in the Driver section. Within the initialization process, the following parameters are set: Driver Context, Link, the Bus Controller ID, as well as a callback function which is used to handle unpairable input packets.

Setting the OnPacketProcessorStatusChanged event provides developers the possibility of notification from Packet Processor and Link about the status of connection. If the connection is interrupted, a developer receives notification about it through this event.

Let's now have a closer look at packets.

The following section will explain how the packets are processed – sent to the device, what are different categories of packets from the point of view of packet flow, as well as types of output packets from the perspective of awaiting a response.

Packets in the Packet Queue can be processed in three different ways.

The first possibility is to define a standard output packet. It's a packet which is sent once, in standard priority. It is placed at the end of the queue, and when its turn comes, it is sent.

The second possibility is to define a so-called periodical packet. It's a new feature in Driver programming that significantly saves the developer's time. It allows registering a packet once, which will then be repeatedly executed in communication with the device, until the communication is terminated. This type of packet is sent, processed, and placed back in the queue.

Initialization of such packet in the source codes is, for example in the case of a Central Unit, sending the packet of type GetTimeRequest for monitoring time in the device.

The third possibility is to define a priority packet. It's a one-time executable packet that because it is marked as priority, gets queued right behind the packet currently being handled, and will be immediately sent for processing as soon as the currently handled packet is finished. Typical packets of this type are user commands.

A priority packet is used, for example, when we want to send activation or deactivation of the output.

The way how Packet Processor handles the packets depends on the particular type of the communication protocol of the external system. From this point of view, we distinguish two categories of packets - output packets and input packets.

Output packets are created by a developer within Device Logic. An example of such packet is SetOutput packet. Output packets are inherited from IOutputBinaryPacket. Each output packet must contain function GetRawBytes, responsible for transforming the packet's data into a binary form.

On the other hand, input packets are created by the device. They can either come as a response to the questions sent to the device, or the device can send them by itself. An example of such packet is OutputStatusResponse packet. Input packets are defined by the IInputPacket interface.

In Packet Parser, data coming from external sources are inserted into input packets, and transformed to the format usable within the internal Logic of the entire Driver.

We distinguish two types of output packets. They can either be waitable, which means that they are waiting for a response from the device; or non-waitable, which means that Packet Processor doesn't await any response from the device after sending them.

Now, we will describe each of them in more detail.

When a packet awaiting a response arrives from the Device Logic, the Packet Processor sends it to the device, while keeping it in the Packet Queue with a set Timeout for response. The Packet Timeout is set in the parameters of the Driver on the Bus Controller, and indicates the timeframe within which the input packet should return.

If it indeed arrives within this timeframe, and there is a packet in the queue that was sent and is waiting for a response, the Packet Processor pairs the input packet with the original output packet. To be able to pair the packets, each output packet awaiting a response contains information about which part of the Device Logic created and sent it, and where the response should be sent back to. This is known as a callback.

After the packets are paired, they are sent together for processing in the Device Logic, and the waitable packet is removed from the queue.

As an example, we can mention periodic inquiries about time on the device, when the response comes to the callback OnGetTimeResponse, where it is processed accordingly.

GetTimeResponseContext is defined as the last parameter of the output packet, and it is the Context which we will see later in the callback. It's an object created by the developer to be able to pair the sent information with the received.

Here we can see the definition of the callback OnGetTimeResponse, where the input packet and responseContext come as parameters, and are subsequently processed within this callback.

There can be a situation that we send a packet awaiting a response, but there is no response within the range defined in the Packet Timeout parameter on Bus Controller, even after the third attempt. This type of connection breakdown at the Packet Processor level usually indicates a device malfunction.

In such cases, a developer is notified of the communication breakdown by calling the function OnPacketProcessorStatusChanged. At the same time, Packet Processor sends a log to the C4 System. Packet Processor itself doesn't attempt to restore the connection, it just informs the developer about it, whose task is to program how to restore communication after its breakdown.

OnPacketProcessorStatusChanged event, which is called in the case of connection issues, is activated during the initialization of the Driver. Implementation of this event is the developer's task, because each device approaches such situation in a different way.

In certain situations, such data need to be sent to the device, to which the device doesn't reply back in any way. We don't expect any response from the device. In such cases, the output packet is sent and immediately removed from the queue.

Some types of devices have their communication protocol designed in such a way, that the device sends certain types of information on its own. The Packet Processor itself cannot determine where to send input data for which it has no matching output data. Therefore, it processes them in a way that all such unexpected packets from the device are sent to an UnhandledPacket callback. Then it's up to the Device Logic developer to decide what type are these data and how to further process them.

Here we can see the OnUnhandledPacket callback, located in the Driver section.

The initial part of the lesson about Driver development covered theoretical background, the second part was focused on the Link module, and in the third part we explored different categories and ways of processing the packets in Packet Processor. This lesson thus provides us with all the information necessary for integrating the device at a standard Gamanet level.

If we aim for a full-scale integration of the device, the next lessons will give us insights to the implementation of other types of Plugins, as well as how to combine them to enable full-featured device configuration.