Interaction with the Packet through Unique-to-packetC Capabilities

The packet can be seen as either one of the simplest data elements in the language or one of the most complex, and hopefully both in a good way. In its simplest form, the packet is an array of bytes, as defined in cloudshield.ph, it simply looks similar to the statement below:

```cpp
//==============================================================================
//  Packet Type
//  Each system may have a slightly different constraint on the buffer for each packet. The typedef below defines the $PACKET for the system.
//==============================================================================
// typedef byte $PACKET[9 * 1024 - 1];
```

Looking at the packet in a different way, it is comprised of multiple protocols layered one inside another. Each layer not only prescribes information about the next layer enveloped inside but also has complex definitions on the construction of the layer itself. A simple HTTP web page request over the simplest Ethernet II link has four major layers to the packet with almost 100 fields of interest. The packet is represented in packetC as an array of bytes with descriptors providing a means to break the array into headers with the individual fields to accomplish a notional network view of the packet as in Figure 10-1. To work with the packet inspecting and manipulating its construction without insight to the packet's construction can seem a bit daunting.

![Figure 10-1. A WAN packet viewed as portions of a byte array](image)

Fortunately, packetC introduces many different tools to work with the packet, in a form consistent with a network engineer's perspective, in order to simplify the coding of a program managing streams of packets. Not only has packetC offloaded entirely the receipt, buffering, and management of the
transmission of packets, but it also provides a number of pre- and post-processing capabilities that provide an understanding of a packet before the first line of code is executed with information passed to the program through the pib and sys data structures. Descriptors introduce yet another major advancement, allowing any of the 100 fields for the packet described above to be addressed by name as if they were simple structure fields, even when one packet to the next changes the byte offset within the packet for those fields.

```c
byte b;
b = pkt[35];

int x;
x = (int) pkt[0:3];

struct BaseType {
  ...;
} myStruct;
myStruct = (BaseType) pkt[36:36+sizeof(BaseType)-1];
```

```c
pkt[j].delete( sizeof( TcpProtocol ) );
```

```c
byte barr[4] = { 6, 7, 8, 9 };
pkt[16].insert( 5, barr );
```

```c
pkt.replicate();
```

In this chapter, some of the other packet operations specific to packetC are discussed with regard to addressing changing the macro level construction of a packet, such as its size and some interactions with how and when it gets processed.

The current packet passed into the packet module’s main is accessible to the user as a byte array. There are several operations that can be done on the packet. The packet will always be referenced as pkt in packetC and is visible throughout a packet module’s global, packet, and block scopes. Since the packet is treated as a byte array, full array slicing and assignment capabilities are present for the packet as either a source or destination of an operator.

### Get Packet Offset

**packet_offset**

This unary operator takes a single descriptor field as an operand and returns an int value that indicates the offset of that field from the start of the packet. Similar to the offset operator, this operator returns the byte value of the field offset. There are a few differences. First, the offset for the field is not from the start of the descriptor but from the start of the packet. A descriptor includes an expression that is used to calculate the offset of the descriptor from the start of the packet. The result of packet_offset is the addition of this value plus the offset of the field from the start of the descriptor. Given that the values in the pib are referenced to determine the offset of a descriptor change from one packet to another, the value returned by packet_offset must be determined at run time. This is in contrast to offset, which is computed at compile time.

```c
struct TcpProtocol {
  short   sourcePort;
```