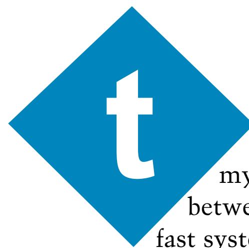
CIRCUIT CELLAR ONLINE

LESSONS FROM THE TRENCHES

George Martin

Upgrading Using Data Packets

Do you ever find yourself between a rock and a hard place? How about between two hard and fast stem requirements? This month, George finds himself with a system that needs to operate as fast as possible, and with as smooth an upgrade to the hardware and software as possible. Sounds sticky, but as always, he makes his way out of it with only a few adjustments along the way.



This month, I find myself caught between two hard and fast systems requirements. The first requirement is that the system operate as fast as possible. The second requirement is that upgrading the hardware and software in the field be as smooth as possible. Sounds familiar, doesn't it?

Let me try to explain the particulars a bit more. My specific system is a communications system, but the issues I discuss in this article can easily apply to almost any embedded system. "As fast as possible" is measured in the response time to user inputs. Imagine a push-to-talk button that keys a transmitter. Any delay in responding to this input will be noticeable. Another example is going off hook and getting a dial tone. Again, any delay here and you will notice.

The upgrade requirements stipulate that the system operate during an upgrade. Yes, you read that correctly. This means that new hardware will be present in systems with old software and vice versa. Also, as new hardware and software become available, new data will flow through the system. Old software needs to not get confused by this new data. And, old and new software need to recognize and communicate with each other.

To get an idea of what this system will look like, imagine several separate complete systems connected by a high-speed network. My problems arise with the data packets that keep these separate systems talking to each other. My hardware uses custom HDLC controllers to ship the data back and forth, but yours might use standard 10BaseT hardware. These data channels are always running, sending fixed size packets full of data when data is available and sending null packets when nothing is available. The transmit/receive buffer size can be configured in the HDLC controller. I selected a large buffer size to place enough data to process in one pass, one that would hold my largest packet (or so I thought). And if my data got larger, I could just split it between two packets. I realize this would require more processing time at the receiving end to piece the data back together.

Also, I need to be able to share data and commands with all the units connected to the network, and these units will be running different versions of the software. My first approach was to define a structure for the packets that went something like what you see in Listing 1.

The *PacketType* variable could be used to identify different data and status packets. I even went so far as to match the layout of the data in the packets with the data layout in my memory. This way, I could use *memcpy* to move the data in and out of the HDLC buffers for transmitting and receiving.

SO FAR, SO GOOD

All went well. As in-house development progressed and changes were needed, I was able to modify the

structures of the data in the packets and keep on trucking. Soon, I needed to distribute new code on a running system. No big deal. I broke the large code file into chucks that comfortably fit into the buffers so that there was little HDLC space wasted. Then, I needed a new structure for this type of packet. There were, of course, all sorts of checksums and other types of error checking embedded in these types of packets, but I'm omitting all of that for this discussion.

So, whenever I had code to transfer, it just became another type of data packet, and I slipped in what you see in Listing 2 every third or fourth packet. This kept the download moving and the loss of throughput in the system is never really noticed. Mind you, there's a lot more to loading code than this, but that's not the problem I'm wrestling with today.

Life was good. I was shipping systems, and changes were frozen except for defect fixes. Next came a new interface card. Whenever these cards were installed a new type of operation was required and a whole new set of data needed to be shared. The software code version that was in the field is called Version5-*xx*. The new hardware and operating software will be named Version6-*xx*.

I designed this Version6-*xx* software to operate with the new hardware, ignoring the upgrade requirements. While creating Version6-*xx*, I created a new packet. This new packet had just a few minor changes for the new hardware. But changes they were, so I created a new structure (see Listing 3).

I took this approach because of a gut feeling that putting the new data in a new packet would slow down the system throughput. The new packet would have little data and would need to be inserted as often as the other packets. Therefore, if I had three regular packet types before Version6-*xx*, I would now have four, or a 25% reduction in throughput. That's the reason why I designed a new packet type.

BE CAREFUL WHAT YOU DESIGN FOR

Next, I started testing the procedure to upgrade in a full system. I started with Version5-*xx* running everywhere, and as soon as I loaded some of the new code of Version6-*xx* a funny thing happened. Version6-*xx* could receive Version5-*xx* packets, but Version5-*xx* couldn't receive new Version6-*xx* packets. At first I was surprised, but after a few seconds of careful thought, I realized that this was in fact the way I designed it to work. However, it's not how I wanted it to work. Oh, well.

The first fix I looked at was putting only the new data into a new type of packet. Two problems arose. First, the new data was tightly coupled with the old data and separating it out would destroy all the clever *memcpy* techniques I was using. Also, the new data

Listing 1—*My original approach can be seen in the packet data structure seen here.*

```
Struct FirstDataPacket {
   INT16     PacketType;    // 1 for the first type Idefined
  UINT16     PacketNumber;
   INT16     VersionOfMySoftware
   INT16 Data1
   INT16 Data2
   INT16 Data3
   INT16 Data4
   INT16 Data5
   INT16 Data6
   INT16 Data7
……
   INT16 DataN
}
```

Listing 2—*This program download packet is what I slipped in every third or fourth packet*

```
Struct FirstDataPacket {
  INT16      PacketType;    // 2 for a software download
  UINT16     PacketNumber;
  INT8 Byte1
  INT8 Byte2
  INT8 Byte3
  INT8 Byte4
  INT8 Byte5
  INT8 Byte6
…. .
  INT8 ByteN
}
```

Listing 3—*Not much of a change, but nonetheless, a new packet.*

```
Struct FirstDataPacket {
  INT16      PacketType;    // 3 for this type
  UINT16     PacketNumber;
  INT16      VersionOfMySoftware
  INT16 SlightlyDifferentData1
  INT16 SameOldData2
  INT16 SameOldData3
  INT16 SameOldData4
  INT16 SameOldData5
  INT16 SameOldData6
  INT16 SameOldData7
……

  INT16 SameOldDataN
}
```

Listing 4—*In this data packet, smaller data types are used.*

```
Struct FirstDataPacket {
  INT16    PacketType;      // 4 for the this type
  UINT16   PacketNumber;
  INT16    VersionOfMySoftware
  INT16    TypeOfDataThatFollows
                            // 1 means 'Version numbers' follow
  INT8     LENGTH          //  8 bytes of data
  INT16    Data1
  INT16    Data2
  INT16    Data3
  INT16    TypeOfDataThatFollows
                           // 2 means 'some other information'
                              follows
  INT8     LENGTH          // 4 bytes of data
  INT16    Data1
  INT16    Data2

}
```

was small, so putting it in its own packet would waste 98% of that packet and, thus, slow down system throughput. The HDLC controller would pad out the packet with null packets and take valuable network time to send them.

The next approach I looked at was to make the blocks of data in the packet smaller and give each its own description (see Listing 4). The benefit of this approach is that I can add and subtract data from this packet type. Old software will not recognize new types but just keep on looking through the packet for types that are recognized. New software can insert data and receive data without needing a completely new packet type defined.

But, the overhead for processing this structure grew dramatically. In Version5-*xx*, I just needed a few comparisons. If they passed, I got right to my *memcpy* routines. In Version6-*xx*, I need a test for every type of data I encounter. Also, the TypeOfDataThatFollows and LENGTH flags were overhead in the buffer. After I made small types of data, I almost had 50% overhead in the packets. With these small sizes, I could easily keep straight what data was needed between the versions, so processing was easy but slow.

So now you see the rock and hard spot I found myself between. I also had the feeling that if I tried to reach a balance between these two approaches I would have the best of neither, and the next change would cause the house of cards to come crashing down.

YOU JUST MIGHT GET IT

After a couple of days of pondering, I realized that the only thing missing was that Version5-*xx* could not read Version6-*xx* packets. If I implemented that function in Version5-*xx*, then the upgrade could proceed. If Version5-*xx* could handle the data in the new packets, then it could extract the data that it could understand and throw out the rest. This approach seemed strange, but as I thought it through, I couldn't find any fault with it. In fact, the routines that process the new packets were already written in Version6-*xx*, so it was just a cut-and-paste exercise.

The upgrade procedure went as follows. All units ran Version5-*xx* software with no new hardware present. Then, I upgraded all units to Version5-*xx* software so it understood Version6-*xx* software packets. The units were upgraded with Version6-*xx* software next. And, the last step was installing new hardware. The operation was a success. There's a bit to learn about code size for different packet types, but the upgrade is easy.

Now, let me tell you the rest of the story. The customer for this product is the U.S. Government. During the installation of a system in the Pentagon on September 11, 2001, all of the technicians were killed in the terrorist attack. The OEM I'm working for was generous enough to offer a replacement system at no cost if the government approved. The government accepted the offer. I rushed these and other changes into the new software to help in the installation and maintenance effort. It's a small story, but one I'm happy to have been a part of.

George Martin began his career in the aerospace industry in 1969. After five years at a real job, he set out on his own and cofounded a design and manufacturing firm. Typical systems that he designs include servo-motion control, graphical input and output, data acquisition, and remote control. George is a charter member of the Ciarcia Design Works Team and most recently, he's been working on the people-tracking system for Bill Gates' new house. You can reach him at george.martin@worldnet.att.net

Circuit Cellar, the Magazine for Computer Applications. Reprinted by permission. For subscription information, call (860) 875-2199, or www.circuitcellar.com. Entire contents copyright ©2001 Circuit Cellar Inc. All rights reserved.