CHAPTER

Packet Optical Networks and Forward Error Correction

4

WHAT YOU WILL LEARN

In this chapter, you will learn more about packet optical networks, the latest optical technology used to link nodes in the Illustrated Network. We'll introduce coherent optical transmission networks and dense wavelength division multiplexing. Then we'll examine forward error correction (FEC) in depth, starting with simple Hamming codes and working up to Reed–Solomon block codes and new standards used in Optical Transport Networks (OTNs).

You will learn how to configure FEC codes for various speed links used in the Illustrated Network.

When it comes to technology, packet optical networks and links depend on three key technology concepts:

- Optical networking with dense wavelength division multiplexing (DWDM)
- Tuneable wavelengths and coherent communication using higher-order modulation
- Forward error correction (FEC).

Although not listed separately, packet optical networks often include interface cards that can be inserted directly into a router or switch. These cards give the device access to bit error rate (BER) statistics and other network parameters previously available only to other network devices such as external transponders or "muxponders."

Although all three of these technologies can be used independently, packet optical networks benefit greatly when they are all used together. Multiplexing has been used since the early days of networking, and fiber optic links use it as well. Many optical fibers are engineered to do more than carry one very fast serial bit stream. The bandwidth available on these types of fiber optic cables, just as certain forms of copper cables (especially coaxial cables) can carry more than one serial bit stream. In copper networks, various channels are distinguished by frequency, but in optical networks, it makes more sense to distinguish the channels by wavelength. Various wavelengths can be multiplexed onto a single strand of fiber—and demultiplexed at the opposite end of the link—with a process known as wavelength division multiplexing (WDM). If the separation of wavelengths is

The Illustrated Network. DOI: http://dx.doi.org/10.1016/B978-0-12-811027-0.00004-7

narrow enough and the resulting channels are dense enough (and there are at least eight channels on the fiber), the result is known as dense wavelength division multiplexing (DWDM). In this system, "non-dense" WDM is known as coarse wavelength division multiplexing (CWDM).

Today, DWDM is standardized for international use as part of the Optical Transport Network (OTN) defined as G.709 by the International Telecommunications Union (ITU). Operation of these advanced fiber optical networks is tied to the use of FEC codes to enable higher-speed connections over extended distances.

Why are FECs needed? Because with this increased DWDM bit-carrying capacity comes an increased risk. A failed or marginally operating link can threaten not only the loss of one stream of bits, but many bit streams that flow on the same physical link. A failed link carrying many gigabits of information can be catastrophic for a network unless some method to compensate for these losses is used. These methods include not only FECs but also fast traffic reroute.

In the sections that follow, one other aspect of packet optical networks is important to keep in mind. First, these networks, as the name implies, are in a very real sense "optimized" for the transport of IP packets. This is not a literal optimization in the sense that IP packets are favored over other types of traffic—optical networks are intended to be versatile. But it is nonetheless true that most voice and video today is carried inside IP packets, and optical networks have to acknowledge that dominance.

However, the second point is that IP packets are usually sent over these packet optical networks inside Ethernet frames. Ethernet has become more or less the *de facto* standard for interface input and output, and packet optical networks are not an exception. A lot of hardware and software exists to generate and interpret Ethernet frames and so it makes sense to use this capability rather than invent yet another Layer 2 frame format (Layer 2 frames are not the same as transmission frames, as we will see again in this chapter).

PACKET OPTICAL NETWORKS AND ERROR CORRECTION

Whole books have been written on packet optical networks, so we'll have to limit this chapter to one aspect of packet optical network or another. We could look at coherent optics with phase considerations, or how optical interface cards make possible for routers and switches to directly monitor the state of the link and protection switch long before a failed link loses gigabits of data. But this chapter will focus on one of the most critical aspects of modern packet optic transmission: the use of FEC to vastly improve the performance and reach of a fiber optic link.

Even in this day and age, all communication networks deal with errors. Errors can be induced by a number of things and showed up in digital networks as individually flipped 0 and 1 bits, bursts of lost digits altogether, and outright loss of

signal until some intervention takes place. Odder sources of error include aircraft landing on seldom used runaways (microwave), nuclear radiation in power plants (fiber optic cable), and even fire heating wires strung over a building (twisted-pair copper). One of the most common causes is inadvertent dig-ups (sometimes called "backhoe fade"). Whatever the cause, all types of media are vulnerable to the loss of one or (many) more bits in a data stream.

There is no Nobel Prize in networking for one very good reason: it's too easy. Only the simple stuff has any chance of working in the messy and complex real world. The only real complication is the need to sometimes deal with mathematics, and mathematics makes many people's eyes glaze over, mainly because they have been taught mathematics in the most inefficient way: as a series of memorized operations and relationships, methods that often prevent learners from seeing the profoundly important connections that numbers have with themselves and the real world. The notations used (the "formalism" of mathematics) can be daunting as well. One famous mathematician (Poincare) supposedly remarked that while writing and poetry involve "giving different names to the same thing"—think of all the different words we have for the color "red"—mathematics involves "the art of giving the same name to different things"...the equal sign rules everything.

Nevertheless, important mathematical concepts can be conveyed and explained without heavy formalism. It just takes a lot more work. But the effort is always worthwhile.

Returning to the simplicity of networking, it is easy to see that networks implement a very basic form of communication. The terms used here were first used by the man who invented in all back in 1948: Claude Shannon. His model starts with this basic form of signaling:

Source/sender/transmitter → channel (noisy, error prone) → destination/receiver

The noisy channel introduces errors. If the signal is analog, meaning that all possible values that the channel can carry are valid (as in an analog voice network), there is no possible way to distinguish error from source signal. Fortunately for analog voice networks, our brains do a good job of filtering out errors in the form of hiss or pops, unless the noise exceeds to power of the original signal (and we all know this happens). On the other hand, digital signals, usually a stream of bits (0 or 1 are the only valid values), offer a way to detect and correct errors on a noisy channel, even if a bit sent as a 1 shows up as a 0 and vice versa. And we're going to see how that happens in a modern network with packet optical links.

This is a good time to be more specific about the sources of error on the communications channel. The nice thing about digitizing everything is that no matter what type of noise is encountered on the link, the receiver can "clean up" the signal and eliminate errors. Seriously, methods explored here can essentially make errors go away, no matter where they came from.

Modern networks do such a good job of dealing with errors that people seldom realize that error handling is always a two-step process: error detection

and error correction. But an elaborate system for error detection and correction is not always needed in all communications systems. For example, receivers of voice and video streams are very tolerant of transient errors, which show up as pops in your ear or "static" on a screen. Data is more sensitive and usually handled with retransmission requests. Some data, such as instructions sent to deep space probes, simply cannot be repeated and yet must be received and used without errors.

The extreme example of critical data that cannot be retransmitted led to the deployment of FEC codes. Initially enormously expensive to implement, all forms of communication can benefit today from a variety of available FEC techniques. A functional definition of FEC could be "the ability of the receiver, based on the possibly flawed information sent by the transmitter, to detect and correct digital errors, within some limiting parameters, without the need to request a retransmission."

But exactly how do these various FEC techniques work? How do they help eliminate errors in modern packet optical networks? And do you have to be a mathematician to understand how these FEC codes are implemented at the transmission frame level? (Fortunately, the answer to this last question is "No.")

What follows is intended for a general reader conversant with modern Internet terms and technology. No math skills are needed beyond an understanding of the bitwise manipulation XOR. In fact, this piece includes the only explanation of FEC codes for the nonengineer or mathematician that I have ever seen. (*Full disclosure*: I couldn't find one, so I wrote my own in 1994.)

PACKET OPTICAL NETWORKS AND THE OPTICAL TRANSPORT NETWORK

FEC is an essential part of a new set of standards that are part of the overall architecture of a Packet Optical Network. At its simplest, this is a network that is optimized for the sending of IP packets, not analog voice or asynchronous transport mode (ATM) cells, over a fiber optic link. The distinguishing characteristic of a packet optical network is that in many cases, but not all, the optical interfaces are not in separate pieces of equipment, but implemented on simple line cards that are placed in the router or switch like any other physical interface.

Packet optical networks depend on more than just new optical interfaces. They also rely on new optical transmission capabilities using higher-order modulation. At the most fundamental level, contemporary optical links can modulate the light on a span with more than just simple pulse amplitude modulation (PAM), where a received signal above a given threshold signals a 1 bit and everything else is a 0. Coherent detection is a receiver-specific technique to allow reception of these newer signals. DWDM transmitters transmit on a specific wavelength, while

receivers can be broadband (that is, they can receive several wavelengths) or wavelength-specific (as in the case of coherent reception).

Packet optical networks can optimize for IP in many ways, but most follow a set of international standards known as OTN. OTN defines standards that are used in packet optical networks to assure interoperability and establish a level playing field for vendors.

STANDARDS FOR PACKET OPTICAL NETWORKS AND FORWARD ERROR CORRECTION

The ITU issues standards (technically, "recommendations" but they have the force of international law) for all aspects of telecommunications. The main for packet optical networks is G.709, which defines a standard OTN. There are also closely related standards such as G.707 for submarine fiber optical cables and G.975, which contains recommendation for FEC code use in DWDM submarine optical systems. However, the FEC codes in G.975 can be used in other places as well.

What should an enterprise or network operator do if their local communications carrier does not offer fiber links that conform to OTN standards? You could always run your own fiber, or course, but in many cases that is not possible.

Fortunately, the IETF has standardized RFC 6363, which defines how FEC can be used with a stream of IP packets, whether the carrier employs packet optical network technologies or not.

When we configure FEC on our core routers at the end of this chapter, we'll see what types of parameters are usually configurable by network operators, even though everything is based on OTN and packet optical standards.

HANDLING SINGLE BIT ERRORS AND BURST ERRORS

There are two categories of errors when bits are stored or sent over a communications channel. There are *single bit errors* and *burst errors*. Single bit errors are likely to occur when bits are stored in memory or on a DVD and need to be read by an application. Scratches on a DVD can reflect the light of the reading laser and flip a bit, and radiation in the form of cosmic rays can flip a bit in computer memory. For communications channels functioning at high bit rates, even the smallest interference can wipe out multiple bits. In either case, reading or sending, FEC codes make the equipment more reliable.

We will examine each type of error here and the types of FEC used to correct bit errors. However, the emphasis will be on techniques used in high-bit-rate fiber optical communication systems such as a packet optical network. Every network, including packet optical networks, are subject to noise on the channel link, after encoding and before decoded the bits. This is shown in Figure 4.1.

Let's look at a bit stream example to see how adding an FEC code can make errors all but disappear. Consider a stream of 100 bits send from a source to a

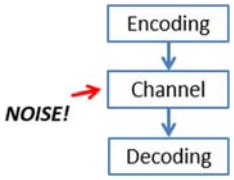

FIGURE 4.1

Channel and noise.

destination. There is no obvious pattern to the source bits, and they can represent many things: a data file, a musical composition, or a video with sound track.

Suppose that the BERT in the line is 5%, or 0.05, or 1 in 20 (all are the same). Now, this is a much higher BER than experienced in all but in the most extreme conditions, a lot of errors will help to illustrate the points being made. In this case, 5 out of the 100 bits sent will arrive with errors: 0 instead of a 1 bit, or a 1 bit instead of a 0. These errors can be spread out as single bit errors, or concentrate in a single burst of 5 errors in a row.

What can be done to our string of 0s and 1s to improve the BER of 0.05 or 5%? The first thing to notice is that this also means we have a "success rate" of 0.95 or 95%. . .but we'd like to make that higher. We can do this by adding some "parity bits" or "check bits" to the data stream and sending not a single bit at a time, but a combination of data bits and check bits (we'll call them check bits and not parity bits because "parity bit" has a more narrow definition in asynchronous communications channels).

Now watch what happens when we add a check bit to every data bit, sending two bits instead of one, but doubling the number of bits needed to represent a message (or halving the true data rate, whichever way you'd like to view it). There are still only two valid "code words" to our system:

0 is sent as 00
1 is sent as 11.

Naturally, errors can flip one of more bits as the move along the channel. Two valid states are sent but can be received as four possible states (that's all that are possible with two-bit code words):

00 (valid for 0)
01 (an error)
10 (an error)
11 (valid for 1).

Note that burst errors can easily change 00 to 11, which is still an error in the data stream, but a valid code word at the line or channel level. These are *undetected errors* and will be passed through as if they were valid symbols in the system. We'll have to do something else later to try and address these.

But if we restrict ourselves to considering single bit errors, and receive either 01 or 10, we *know* there has been an error on the line because we never send 01 or 10. These are not valid code words.

HAMMING DISTANCE AND HAMMING CODES

So we have *detected* the errors (in this case). But we can't correct them, even knowing they are in errors. Should 01 be "corrected" to 00? In a mathematical sense, 01 is "closer" to 00 than it is to 11, as 10 is "closer" to 11. But not really: either bit error pattern is certainly equally possible, and it's just as likely 00 would become 10 or 01 or that 11 would become 10 or 01 (Figure 4.2).

Now, our simple two-bit code words can detect some errors, but correct none. And at the cost of 100% overhead, we'll have to do better. What if instead of two bits for each data bit, we sent *three* bits? (I owe this example to Professor Benjamin Schumacher at Kenyon College.)

Now there are eight possible code words, but we use only two valid ones, just as before:

000 (valid for 0 bit sent)
001 (an error)
010 (an error)
011 (an error)
100 (an error)
101 (an error)
110 (an error)
111 (valid for 1 bit sent).

Now let's look at a single bit error in one of these "triplet" code words. What if 000 is sent, but an error causes it to be received as 010? We have detected the error, just as before. But should the error be corrected to 000 or 111? Now we do have a case where the distance between errored word and valid word is closer to one valid code word than the other. 010 is "one step" (one bit change) away from 000, but two steps away from 111, meaning we would have to change two bits to create the valid 111 code word. This is called the *Hamming distance* and we should always choose the valid code word closest to the received error to correct the error.

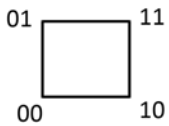

FIGURE 4.2

Simple two-bit code as square.

With burst errors, however, we might make things worse instead of better by using the Hamming distance. What if 111 is sent, but a burst flips two bits and we receive 001? We would "correct" this error as 000, and in doing so *add an error* to the bit stream instead of subtracting one. This is a hazard of all error-correcting codes: in some circumstances, we actually *increase* the net BER when trying to correct errors. What we have with our simple three-bit code is a simple form of SECDED: single error correction and double error detection.

As long as most of the bit errors are single bit errors, a SECDED system is fine. (Later, we'll see that some codes can use a *soft-decision* method to add some measure of confidence to the simple *hard decision* "is-it-0-or-1?" code implementation.) For example, if all previous errors have been two-bit bursts, then 001 could use a soft decision implementation to correct the code word to 111 instead of the hard decision 000.

Leaving hard and soft decisions aside, we should always correct the error by using the minimum Hamming distance from valid source code word to errored received code word.

Often, the eight states of our simple three-bit code are laid out in cube, with all 0xx code words forming one square and all 1xx code words forming the other. Any errored state will always be fewer "edge steps" away from one valid code word than the other (Figure 4.3).

The Hamming distance (d) between the code words used in any FEC system is important. If $d = 1$, then we essentially have only 0 and 1 to work with, and any bit error at all will turn one into the other—not a good situation. If $d = 2$, as in our simple "send each bit twice" code using 00 and 11, then things get more interesting. As we saw, we can detect errors in the form of 01 or 10, but we can't correct them. We call this a (2,1) code, where each information bit (1) is sent as two-bit long code words (2).

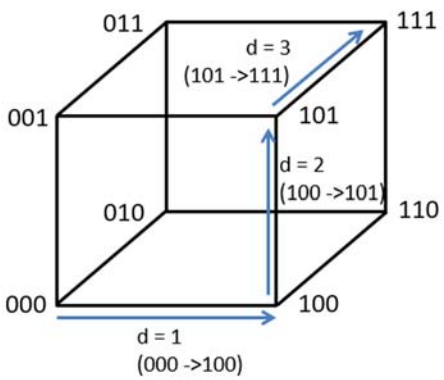

FIGURE 4.3

Three-bit code Hamming distance cube.

FIGURE 4.4

Correction and detection with Hamming distance = 4.

Table 4.1 The Relation Between Hamming Distance and Error Correction Capabilities

If *d* is	Then corrects this many	And detects this many
1	NA	NA
2	NA	1
3	1	
4	1	2
5	2	
6	2	3
7	3	
8	3	4

But when $d = 3$, and the code is in the form (3,1), this allowed us to correct any single bit error, as we saw, because any single bit error (see our list) still leaves us closer to one alternative of the other (000 or 111). We can detect multiple bit errors, but we can't correct them (Figure 4.4).

Generally, if the Hamming distance d is equal to some even number x, then the Hamming code can detect $d/2$ errors and correct $(d/2) - 1$ errors. For example, if we chose code words so that the Hamming distance $d = 6$, then we can correct $(6/2)-1 = 2$ bit errors and detect $6/2 = 3$ bit errors. This is shown in Table 4.1.

A BETTER HAMMING CODE METHOD

Note that this higher performance is achieved at the price of 200% overhead (or three times the bandwidth required).

In 1948, Claude Shannon proved that enough information could be sent through a noisy channel to allow error correction. But the invention of a system to actually perform the error correction came from Richard Hamming, who worked with Shannon and even shared an office with him for a while. Shannon included Hamming's work as an example in his original paper.

The Hamming code investigated here, which is still in use in computers and cellular systems, adds not 200%, but about 43% overhead to the information we are sending. In order to do this, we have to get a lot more inventive with bit manipulation than just sending more copies of the same bits. We have to add a bit of processing to the bits before they leave and as they arrive. This can add

delay to the throughput of the system, so different FECs are often used in different places and for different applications.

For memory reads from a computer, 43% overhead is not too bad. But in case you're wondering, modern FECs for packet optical networks add anywhere from 5% to 20% to the raw data stream because the added delay is worth the enhanced performance of the link. As always, the trade-off is in processing power and throughput delay (Figure 4.5).

The popular Hamming code used here is written as (7,4). This means that the sender looks at four data bits at the time (note we have progressed from examining 2 bits at a time, to three, and now four, and at each step we get the ability to do more and more). Imagine is we could deal with thousands (!) of bytes, not bits, at a time. . .we'll do that soon.

Each group of four bits is examined and to them three check bits (also called parity bits) are added, giving the "7" in the code notation. These are always the important numbers in FEC notations: how many bits total are we sending, and how many of them are raw data bits? For the (7,4) Hamming code, the answers are 7 and 4.

We can add these "Hamming numbers" to Hamming distance and create a list of essential parameters for all error correction techniques. Later, we will apply them to byte-oriented Reed−Solomon codes, but for now, we'll stick to bit-oriented Hamming examples. Here are the parameters:

- m or s: Some texts use m and some use s, but to avoid confusion, we'll just use the term "symbols" to indicate whether the code is acting on individual bits, 8-bit bytes, or even some other unit to produce the code words sent over the wire. Technically, this is the "smallest correctable entity" (usually bits or bytes) that the code can operate on.
- n: This is this number of symbols in the overall code word produced by the method. In our example, $n = 7$.
- k: This is the number of information bits that are taken as a group to produce the code word. In our example, $k = 4$.

FIGURE 4.5

Where error correction fits in.

- *d*: This is the Hamming distance that we have already talked about. In our example, you have to change 3 bits to make another valid code word, so $d = 3$.

There is one other parameter, *t*, that we'll introduce later, when we talk about Reed−Solomon codes. For now, all we have to know is that *t* is related to the difference between *k* and *n* (i.e., $n-k$) and measures the capabilities of the number of check or parity bits that are added to the information bits *k*.

HAMMING CODE IN ACTION

In this example, we'll call our 4 data bits d_1 through d_4, and the added parity bits p_1 through p_3.

It's obvious that the values of the four data bits we start with are provided by the source. So how do we decide what values the three check/parity bits should have? It's clear that the 4-bit input groups can only be one of the 16 possibilities listed here:

0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111.

(I used to watch people draw these up by columns, right to left, and finally realized that the rightmost column goes 0-1-0-1-0-1… and the next column goes 0-0-1-1-0-0-1-1… and third goes 0-0-0-0-1-1-1-1… for however many you need. Who does them horizontally?)

To derive or compute the value of the three parity bits, we use the bitwise exclusive-or (XOR) operation: the bits are 0 or 1, but not both. So 0 XOR 0 = 0 (false), and 1 XOR 1 is also 0 (false) because both values cannot be the same with XOR. On the other hand, 1 XOR 0 = 0 XOR 1 = 1 (true) because it's one or the other, but not both.

With these simple rules, we can formulate three rules for determining the value of the parity bits:

$p_1 = d_1$ XOR d_2 XOR d_4
$p_2 = d_2$ XOR d_3 XOR d_4
$p_3 = d_3$ XOR d_1 XOR d_4

Don't be thrown by the "double XOR" operation: just take the result of the first XOR and use it as one of the values for the second XOR.

At this point, it might be a good idea to tackle the issue of what the numbers *mean*. As soon as mathematics rears its ugly head, in fields as diverse as quantum physics to relativity, we can take one of two attitudes. Either we can "shut up and calculate" and follow the rules because they work, or we can wonder what the numbers are telling us about the way reality works at a more fundamental level.

How can a series of XORT operations reveal whether received bits are different than the ones sent and, moreover, which bits they are? I don't claim to be a mathematical theorist, but way to look at it is to consider how XOR works. If it's "one or the other but not both" then a bit-flipping error to either component reverses the outcome (the XOR'd 0 result becomes 1, or 1 becomes 0). If we overlap these XORs cleverly, not only can we detect the error, but we can find out where in the string of bits the XOR has gone wrong and set the offending bit back to the proper value. It is at once astonishing to the novice and inevitable to the mathematician.

Let's investigate an example using our Hamming (7,4) code. The data bits to be sent are 1010. What are the values of the parity bits?

Well, $p_1 = d_1$ XOR d_2 XOR $d_4 = (1$ XOR $0)$ XOR $0 = (1)$ XOR $0 = 1$, so $p_1 = 1$

In the same way, $p_2 = d_2$ XOR d_3 XOR $d_4 = (0$ XOR $1)$ XOR $0 = (1)$ XOR $0 = 1$, so $p_2 = 1$

And $p_3 = d_3$ XOR d_1 XOR $d_4 = (1$ XOR $1)$ XOR $0 = (0)$ XOR $0 = 0$

We can visualize the relationship between the four data bits (d_1 through d_4) and the three parity/check bits (p_1 through p_3) as overlapping circles (Figure 4.6).

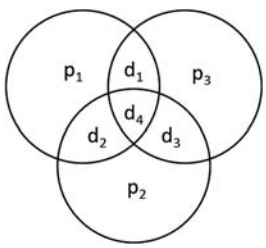

FIGURE 4.6

Data and parity bits as overlapping circles.

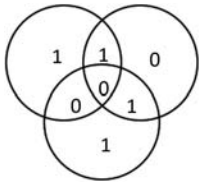

FIGURE 4.7

Parity bit values for the 1010 Example.

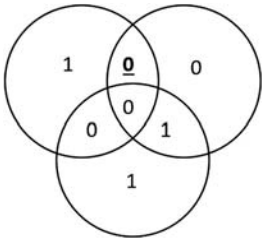

FIGURE 4.8

Correcting a d_1 bit error.

Note the relationship between three of the four data bits and the p-bit that depends on them. If we fill in the data values from our 1010 example and then add the values of the parity bits, we get this result (Figure 4.7).

Note now that each of the "p circles" contains an even number of ones. This is the direct result of our XOR operations and a key to detecting and correcting single bit errors in our example.

Suppose, for instance, that during transmission of our bit stream data bit d_1 is garbled and interpreted at the receiver as a 0 instead of a 1. The result would be as shown in Figure 4.8, where the errored bit is shown in bold and underlined.

How would the receiver know that there was an error in the first place? Let's work through the examples at first as if the receiver actually could look at our circles and make deductions based on their values. Then we'll see how computers actually do it.

If we were the receiver, we would first note that the p_1 and p_3 circles no longer have even parity (i.e., an even number of 1 bits). These two circles now have odd parity. Circle p_2, on the other hand, still has even parity. Therefore the errored bit is where the p_1 and p_3 parity circles overlap, or with data bit d1. We can correct it easily, as shown in Figure 4.9.

Is it really that simple? Yes, it is. Now let's consider a bit error in data bit d_4, which is included not in two parity circles, but in all three. Again the errored bit is in bold and underlined, as in Figure 4.10.

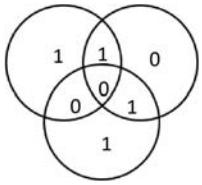

FIGURE 4.9

The d1 bit error corrected.

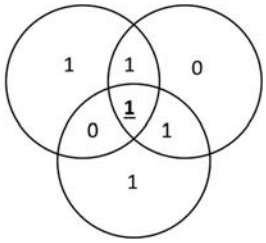

FIGURE 4.10

A data bit d4 bit error.

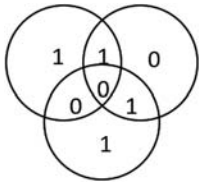

FIGURE 4.11

The d4 data bit error corrected.

Note now that the even parity of *all three circles* is thrown off. All circles have an odd number of 1 bits. Therefore, the error is where all three circles overlap, or in data bit d_4. Again, this is easy enough to correct, as shown in Figure 4.11.

There's only one more case to consider. What about a bit error that wipes out not one of the data bits, but a parity bit? Can the Hamming error protection system detect and correct errors in the parity bits themselves? Yes! Figure 4.12 shows a case where parity bit p2 has been changed from a 1 to a 0 bit during transmission.

In this case, it's not two overlapping circles that violate even parity, or even all three, as in the previous cases. Now, only the p_2 circle has odd parity, so this

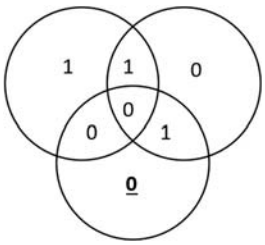

FIGURE 4.12

A bit error in the p_2 parity bit.

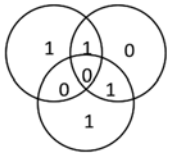

FIGURE 4.13

The p2 bit error corrected.

must be where the error is. When only one circle has a parity error, then it is the parity bit itself that needs correcting, as shown in Figure 4.13.

And those are the only three cases we have to worry about: a parity violation in one, two, or all three parity circles. The only difference would be which circles contain the parity violations. Correcting the parity also corrects the error.

HAMMING CODE IMPLEMENTATION

Now, you might wonder how the receiver board in a computer looks at circles that way we just did. Of course, computers are much too dumb to actually do what we humans just did very easily. But we can get the same result with XORs, just as we did at the sending side.

Here's how. Both sender and receiver know that they are sending and receiving a Hamming (7,4) code (this is a must). The receiver first isolates the four data bits and ignores, for the time being, the value of the three received parity bits. Then the receiver recalculates the parity bit values, exactly as the sender did.

That is:

$p_1 = d_1$ XOR d_2 XOR d_4
$p_2 = d_2$ XOR d_3 XOR d_4
$p_3 = d_3$ XOR d_1 XOR d_4

Once the values of these three p-bits are computed, the receiver performed one additional XOR operation: the receiver XORs the received parity bits with

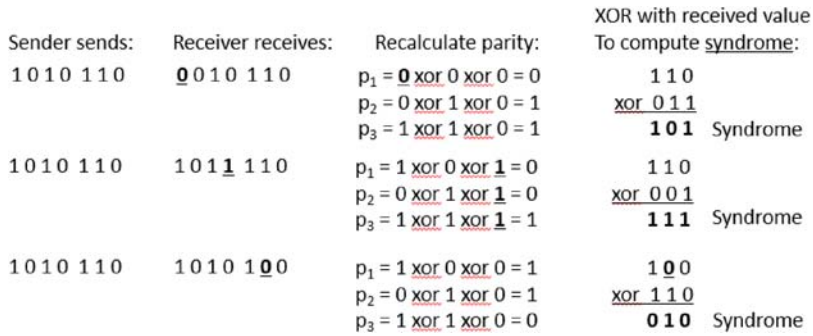

FIGURE 4.14

Syndrome computation at the receiver.

Table 4.2 The Syndromes for the (7,4) Code Example

Syndrome z	000	001	010	011	100	101	110	111
Unflip bit:	All OK!	p3	p2	d_3	p1	d_1	d_2	d_4

the recomputed parity bits. The result of this operation is known as the *syndrome* of the operation. (In many texts, the computed syndrome is called z.)

Figure 4.14 looks at the bit errors in each of the cases examined (i.e., errors to d_1, d_4, and p_2). The first two columns show the seven bits as sent and received, then the third column shows the recalculated parity values at the receiver. The last column shows the syndrome computed when the received parity bits are XOR'd with the recalculated parity bits.

How does the syndrome help the receiver figure out which bits are in error? Simple: you look it up in a table! The three bits of the parity syndrome z form the index of an eight entry tables: 000 to 111. Each possible result tells the receiver which bit is in error (or if there is no error at all, which we want to be the case every time).

Table 4.2 shows the syndrome table for the (7,4) code.

Let's apply each of our examples from our "circle examination method" to our new "table lookup method":

1. The d_1 bit error syndrome = 101, so the receiver should flip bit d_1. (That is, if the received bit is 0, make it a 1, and if the received bit is 1, make it a 0.)
2. The d_4 bit error syndrome = 111, so the receiver should flip bit d_4.
3. The p_2 bit error syndrome = 010, so the receiver should flip bit p_2.

Note that if the received parity bit pattern matches the recalculated parity bit pattern exactly, the syndrome would be 000, a case that occurs much more often than an error, hopefully.

We've looked in detail at only three of the possible eight cases (no errors, to errors in each of the four data bits and three parity bits). I hate to do this, but the "proof" or demonstration that the syndrome method works for all five of the other cases is, as they say, left as an exercise to the student. You will either verify the method or make an unexpected mathematical discovery.

The nice thing about making this table is that it will be the same for every implementation of the (7,4) code and therefore can be hard-coded into the receiver chip. The other nice feature is that computers do many things very quickly, and one of the things is using a computed result as an index into a table. So this method is consistent, simple, and fast.

Another exercise is to examine cases where *two* (or more) bits out of the seven-bit code words are flipped or errored. These can be consecutive or nonadjacent bit errors, it makes no difference. In these cases, it is easy to show that error correction using computed syndrome results not in error correction (i.e., restoral of the original seven bits sent), but in *more* bit errors than we started out with! When a system is designed to correct single bit errors, that is precisely what it does.

BURST ERRORS AND INTERLEAVING

So none of the methods we just examined do anything about the issue of burst errors. Single bit errors are fine to correct, but many errors come not as single events that smash individual bits, but longer bursts of static or other types of impulse noise that wipes out several bits in a row: often tens of bits, sometimes hundreds, occasionally thousands. In cases where thousands of bits might be flipped or simply eradicated, one of the reasons that we have layered protocols comes into play. One of the main tasks of an upper layer is to recover from uncorrectable errors that occur at a lower layer. For example, uncorrectable errors that might occur on a hop between two intermediate systems can be detected and fixed by an end-to-end resend between the client and server at the endpoints.

But maybe we can do something about our error-correcting system to make it more robust in the face of limited burst errors (there will always be a point where the length of the burst exceeds the ability of the designed correction code's to fix bit errors).

The method commonly used is called *interleaving*, sometimes seen as *scrambling*. With interleaving, we do not send the code bits sequentially as soon as we get them. We buffer the bits, both the data bits and derived parity bits, at the sender until we have a set of bits in some predefined structure. Then we can interleave the bits and send them not sequentially as they arrived, but in another pattern altogether.

Let's look at a simple example using of initial (3,1) code which sent every data bit three times, so 0 became 000 and 1 became 111. As we saw, single bit errors could be corrected as 010 -> 000 or 110 -> 111 due to the Hamming

distance characteristics. But if 000 became 110, we lost that ability by trying to correct the string to 111.

But let's buffer five of these code words together first. So if, for instance, 0 came in and became 000, we would not send it right away. Let's call that AAA and wait for another bit like 1 to became 111. That would be BBB in our simple system, with the three letters standing for a particular 000 or 111.

In fact, let's do this five times: AAA BBB CCC DDD EEE. (Note that the spaces are simply for convenience of visualization: in practice, the bits would all run together.)

Not we're ready to send the 15 bits. . .but interleaved. We would send ABCDE ABCDE ABCDE. At the received, even before error correction, the original "frame" structure would be recovered by "unscrambling" the bits. Only after de-interleaving would error correction be applied (010 -> 000) and the bits decoded into the sent information (000 -> 0).

Now, suppose a burst error wiped out four consecutive bits in the stream, perhaps like this:

ABCDE ABXXXX BCDE

When this sequence was un-interleaved, we would have: AAX BBB CXC DXD EXE.

We have just, through the process of interleaved, made four single bit errors out of a four-bit burst error! We have, with this simple method, created a system that can recover from bursts up to five bits long. However, we have also introduced another buffering and processing delay at the sender and receiver, and delay and processing burden that grow with complexity and robustness. But given the delay and burden of resending large chunks of errored data, the trade-off is often more than worthwhile.

In practice, we can interleave much more complex "transmission frames" and actually *nest* the interleavings to create more and more robust burst error correction methods. Audio CDs next interleavings and error correction to create a method that can recover from some *3600* consecutive bits in error. This "delay" is acceptable because audio is strictly a one-way process from disc to your ear. The interleaving delay is absorbed long before the disc is burned, and the delay to de-interleave when you press "play" is acceptable because once the audio pipeline is filled, the decoding is a continuous process.

So a more complete look at the steps in an error-correcting process includes one or more interleaving stages, as shown in Figure 4.15.

MODERN FEC OPERATION

We are getting closer to configuring FEC on our inter-router SONET/SDH links. But no one would use the simple Hamming (7,4) method and process every four data bits before and after they were sent, not on a link that runs at multiple

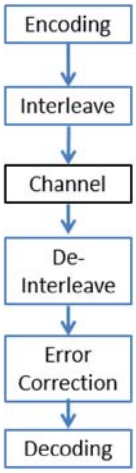

FIGURE 4.15

Interleaving and error correction.

Gigabits per second. For that use, we need something much more robust. Fortunately, something much more robust does exist, in the form of Bose–Chauduri–Hocquenghem (BCH) codes and Reed–Solomon (RS) codes, both invented around 1960. It took a while before they became common, however, because 1960 was long before computer line cards were small enough and fast enough to employ FECs based on either BCH or RS.

At the risk of getting too abstract, we should note that RS codes are a subset of BCH codes, and both are in turn examples of cyclical codes. In fact, there is a family of *linear block codes*, which are codes that act on blocks of symbols, usually byte or octets and not bits. Some linear block codes are *polynomial codes* that are based on expressions like $x^5 + x^2 + 1$. In practice, the polynomial mathematics is handled by simple bit-shifting of the polynomial as a string of 0s and 1. For example, the polynomial $x^5 + x^2 + 1$ is the same as 10011. You just put a 1 in a "1s position" and the "x^2 position" and the "x^5 position." The rest of the bits are 0. These are the strings we use to XOR with our data stream at the sender and receiver.

Some polynomial codes are *cyclical codes* based on Galois fields ("fields" mean "based on modular n arithmetic") that can never yield a result that is not part of "code field" (this is a good thing). Examples of cyclical codes include BCH and RS codes, although the whole field remains a very active place for research.

It took a long time for computing to catch up with the mathematics. When I wrote my first book on ATM cell-relay technology in the early 1990s, ATM used a simple (for today) BCH (40,32) code that covered the 32 bits of the ATM cell header and added 8 parity bits to the 40-bit header error control (HEC) field.

Note that outside of the 5-btye cell header, the rest of the 53-byte cell had no error correction or detection at all! Let the higher layers worry about that.

The BCH (40,32) code for ATM provided SECDED: single error correction, double error detection, all based on the Hamming distance of the code words used.

FEC AND SONET/SDH

In a way, it is unfortunate that our router network uses SONET/SDH links instead of more modern OTN links. We'll see why that is important soon, but for now it is enough to note that the OTN standard (G.709) was finalized in 2012, after the first edition of this book was written. However, SONET/SDH was standardized in 1988 (along with ATM), and there is still a lot of SONET/SDH out there. This section will explore the limitations of employing FEC on SONET/SDH links, and then examine the recommendations for using FEC on the *digital wrappers* established by the OTN standard.

FEC codes used in SONET/SDH do not act on individual bits (the *s* or *m* value for symbols). These BCH and RS codes act on 8-bit bytes or octets. They use multiple levels of nested interleavings, and, as we have seen, each interleave handles error correction for more and more consecutive bit errors by turning them into single bit errors at heart. BCH and RS codes are much more complex than Hamming codes, and we won't be looking into how the mathematics works in any detail. One good reason is that the FEC codes are not in the form (7,4) but have notations like *a shortened form of BCH-3 (8191, 8152)*. This code adds 39 parity or check bits to 4320 SONET/SDH information bits and corrects any 3 bits in error. This method "sprays" the FEC bits around inside a SONET/SDH frame to correct up to 24 bits in error, consecutive or not.

The FEC codes used in SONET/SDH are considered to be usages of *in-band FEC*. This is unfortunate, because all FEC codes are actually "in-band" and sent right along with the information bits they protect. If our Hamming (7,3) code was not in-band, but truly "out-band" then we would have to send the 4 data bits on one channel and the 3 parity bits on another, somehow synchronizing their arrival and pasting them together at the receiver to perform the operations needed to calculate the syndrome. A better name for the techniques used would be "in-the-frame-FEC" for the SONET/SDH method and "out-of-the-frame-FEC" for the others.

For all its technical inaccuracies, the use of the term "in-band" for SONET/SDH FEC usage is not all bad. If nothing else, it points out how unique the FEC technique is for SONET/SDH, and how different it is from "out-band" FEC used for OTN links. The rigid structure of the OSNET/SDH frame means that only higher link speeds have enough "spare bits" and therefore room in the frame for the additional FEC bits. So the official G.707 FEC revision for SONET/SDH applies only to OC-48/STM-16 line rates and above, starting at about 2.488 Gbps.

The rigid structure of the SONET/SDH frame, which defines the precise number, structure, and meaning of bits that must be sent every 125 microseconds (1/8000th of a second), cannot change the line rate at which bits are sent. This means that the FEC definition must be very creative when "retro-fitted" onto a SONET/SDH link. The method is defined mainly for OC-48 or STM-16, but higher rates such as 10 Gbps can be addressed as long as the line rate is a multiple of 48 (SONET) or 16 (STM). You basically interleave N/48 or N/16 bit streams 16 octets at a time in a very complex pattern to place the added FEC bits inside the SONET/SDH frames.

To see where the FEC parity bits are placed in a SONET/SDH frame, recall that the basic SONET/SDH frame is not a linear structure, but a two-dimensional frame consisting of nine rows and a variable number of columns, as mentioned in Chapter 3. The initial columns of each frame are the transport overhead (TOH) and the remaining columns are the payload data bits. Frames are sent row by row, left to right, until all none rows are sent and then the process begins again with the next frame.

At the OC-48 or STM-16 line rate, the TOH consists of 9 rows and 144 columns of octets (followed by 4176 octets of payload, which we need not consider here). For FEC purposes, we assemble a "super-frame" of eight consecutive frames, forming a kind of cubic structure that we can index with a coordinate system such as a, b, c, although with certain twists.

In-band FEC uses only the first 18 columns of the TOH and not all 144 columns. Confusingly, the columns are grouped not into 8-bit octets, but into 16-bit groups indexed by parameter b, which can thus have the value $b = 1$ to $b = 9$ (actually, these 16-bit groups are only for OC-48/STM-16 FEC, but that's all we consider here). Thankfully, the a parameter still indexes the row, so ranges between $a = 1$ and $a = 9$.

What about c? This index does not indicate which frame of the super-frame being considered, as you might expect, but the individual bits within each b group of 16 bits. The maximum value of c varies, and is always N/3 for SONET and N for SDH. At the OC-48 line rate, $c = 16$ and for STM-16, $c = 16$ also. In other words, the value of b is always 9 for all sizes of TOH, but c gets larger and larger. At OC-192 and STM-64, the maximum value of c is 64. Tricky, but not hard to figure out.

So a set of values of $a = 3$, $b = 6$, $c = 9$ means the third row of the TOH of the super-frame, sixth group of "b-groups," and the ninth bit (left to right) of the sixth "b-group." Therefore, the first A2 framing bit in the first frame of a super-frame is indicated by $a = 1$, $b = 4$, $c = 1$, or S(1,4,1) in G.707 notation.

How does the a,b,c system help us to figure out where the FEC bits are placed in a SONET/SDH frame? Well, let's look at the FEC code used first. SONET/SDH FEC is based on a BCH-3 code, shortened from (8191, 8152) to cover 4320 payload bits with 39 ($8191 - 8152 = 39$) FEC parity bits. Now, how can they just shorten the code like that? By realizing that many of the fields in SONET/SDH frames are set to all-zero, you can just take out these 0 bits at the sender and add

them back in at the receiver, you can make the processing times much shorter. The BCH-3 code can correct up to three bits in error. But we are going to interleave these eight times (that super-frame structure), so the result is a method that can correct bursts up to 24 bits in length.

To make sure people are paying attention, these 39 FEC parity bits are referred to in the standard as a_n, where $n = 0$ to 38 (not 1 to 39, of course). And these bits should never be confused with the a vector coordinate in the S(a,b,c) notation, which ranges from $a = 1$ to $a = 9$. This is a good way to make sure instructors and consultants have jobs, but not so good for students seeing this for the first time. Here, we'll refer to these 39 FEC bits as "FEC parity bit a_n" so there is less confusion.

First, break up those 39 parity bits into 3 groups of 13 bits each. Then we can "spray" them around the rows of the SONET/SDH frame they apply to. This is done to minimize the delay: if the FEC bits for a row were not in the row they protect, then more than one row would have to be buffered to compute the syndrome and apply the corrections. This keeps the delay down to about 14 microseconds. There are exceptions: there is no room at all in the 1st and 4th rows of the SONET/SDH frame for FEC bits because the whole row is taken up with other functions and pointers. So the FEC bits for these rows are sent in the following rows, adding a slight delay to sender and receiver processing.

Now we can see exactly where the 39 parity bits are placed in the frame. This is shown in Figure 4.16. Note that each FEC symbol in each row represents

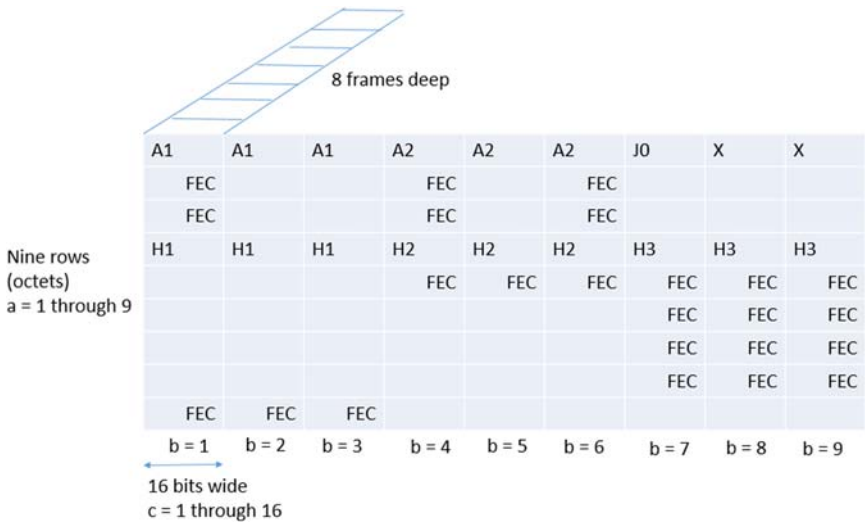

FIGURE 4.16

In-band FEC and the SONET/SDH "Super-frame.".

13 parity bits. Although not shown in the figure, the 13 bits are located after three unused bits in the b group, in bits $c = 4$ through 16.

How does a receiver know if the link is using in-band FEC or not? There is a bit located at S(3,9,3) that, taken with the 7 bits in the super-frame "behind" it, form an 8-bit repeated field called the FEC Status Indicator (FSI). For those who have been paying *really* close attention: yes, this is the last unused bit before the FEC parity bits for row 2 in the last b column. Only two bits are used for the FSI (the last two bits in the super-frame): 00 means "no FEC used," 01 means "FEC in use," and the 10 and 11 values are undefined.

There are more details not covered here, such as how the FEC bits are organized in the row, but this is most of it. There is a reason to cover all this complexity, although the process remains essentially transparent to users. Is all the effort really worth it? Aren't SONET/SDH link already low enough in error rates that we can just leave them alone?

Some vendors say "Yes, leave it be..." Because of all this complexity, not all hardware vendors support FEC for SONET/SDH at all.

Yet there is a tangible payoff. If we assume the errors are independent and Gaussian (very normal assumptions), then the FEC can improve a link with a BER of 10^{-10} to one of 10^{-14}, or 10,000 times fewer errors. Even a seriously degraded link with a BER of 10^{-6} (truly horrible for fiber optic links) improves to 10^{-14} with FEC in use. This can mean the difference between keeping a link in service and having to replace it because of poor performance.

Every FEC has limits too. This method fails at about 10^{-3} BER (1 in every 1000 bits in error). Here the FEC will add to the errors instead of correcting them.

FEC AND OTN

Once FEC moves from the strictures of the rigid SONET/SDH frame structures and line rates, the shackles are off and the sky's the limit. A few simple changes make it possible to use almost any FEC you like, as long as sender and receiver agree on its use. One key change is that OTN standards (often just called "G.709") allow various types of *digital wrappers* to operate at tunable wavelengths. When it comes to what you can do, the rules of OTN are much more permissive than they were in SONET/SDH.

You want to send IP packets over an OTN link? Go ahead. Ethernet frames? No problem. SONET/SDH? Yes, even those frames and rates have been adopted for OTN use.

The three Optical Transport Unit (OTU) levels of the OTN digital hierarchy are shown in Table 4.3.

Table 4.3 OTN and SONET/SDH Line Rates

G.709 OTN level	Line rate	Contents
OTU1	2.666 Gbps	OC-48/STM-16 (2.488 Gbps)
OTU2	10.709 Gbps	OC-192/STM-64 or WAN PHY for 10GBase-W Ethernet
OTU2e	11.09 Gbps	10G LAN Ethernet from switch/router at 10.3 Gbps (G.Sup43)
OTU2f	11.32 Gbps	10 Fibre Channel
OTU3	43.018 Gbps	OC-768/STM-256 or 40G Ethernet
OTU3e2	44.58 Gbps	Up to four OTU2e signals
OTU4	112 Gbps	100 Gbps Ethernet

There are four key aspects of the OTN standards (and I present them in a particular order here):

Protocol Transparency—OTN can carry all types of payload data, including, as mentioned, IP packet, Ethernet frames and SONET/SDH frames. OTN line rates are 7% higher than their corresponding payload line rates. The additional bits are used for additional overhead and to accommodate many different types of FEC methods.

Asynchronous Timing—OTN maps payloads into digital wrappers in an asynchronous manner. This means that the OTN clock generating the frames can run freely and not be locked to the "client" signal clock. Timing mismatches are handled by allowing the payload to float in the OTN wrapper. As an option, the payload clock can be used to generate the timing for the OTN frames.

Management—OTN allows monitoring and management of both link segments and end-to-end. Segments can overlap (a span can be a member of more than one management segment) and the architecture allows up to six monitoring segments at any point. So if a link from Network A passes through Network B, the operator of both networks has access to relevant monitoring and management information.

FEC—OTN does not add FEC onto a basic architecture, but builds FEC consideration into it fabric. As should be obvious by now, very high data rates over very long distances are subject to significant noise and large error bursts. These links could not function without FEC. OTN uses RS (255, 239) FEC code built right into the OTN frame outside of the frame payload area in a special section called the *FEC area* (and so is considered to be *out-band FEC* instead of in-band: it is outside of the payload, but still inside the OTN frame). Each 255 byte block contains 16 FEC bytes generated from 239 data bytes. As the code designation indicates, OTN can correct up to 8 bytes of errors in a block and detect up to 16 bytes of error in a block, and more sophisticated interleaving can improve this at the cost of more processing and buffering delay.

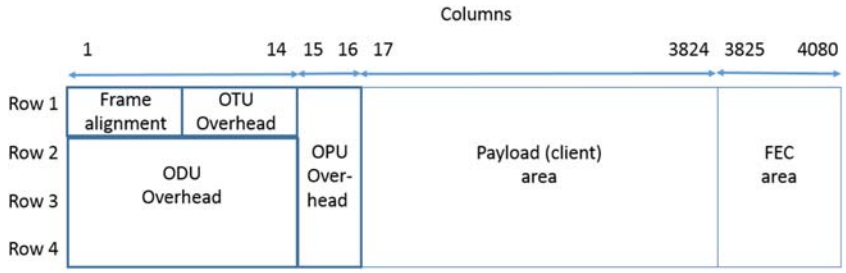

FIGURE 4.17

The OTN frame structure.

THE OTN FRAME AND FEC

Like SONET/SDH, OTN is layered hierarchy which spans the optical fiber at the bottom to the payloads (called "clients" in OTN) that are carried inside the wrapper: IP packets, Ethernet frames, SONET/SDH, and others. It's not necessary to detail these layers, but it will be a good idea to take a quick look at the OTN frame structure, if only to appreciate the difference between FEC accommodation in SONET/SDH and OTN.

OTN frames have 4 rows and 4080 columns of bytes (octets). Frames are sent starting with row 1, column 1, left to right, to row 4, column 4080. Each row consists not of bytes as in SONET/SDH, but 16 interleaved FEC "blocks" of 255 bytes ($16 \times 255 = 4080$). Each 255-octet block consists of 238 bytes of payload, 1 octet of overhead, and 16 bytes of redundant FEC bytes. The FEC bytes are carried in a special "FEC area" in the frame, columns 3825 to 4080 (16 FEC bytes $\times$ 16 blocks = 256 columns).

The 16 blocks in each row are interleaved so that each block can correct up to 8 bytes (64 bits) in error, and burst errors can be corrected up to 128 bytes long. All in all, the frame structure in OTN is "optimized" for FEC and expects FEC to be used, not added on. Unlike SONET/SDH, the FECs used in OTN can be or more than one type, depending on whether the optical span is a submarine cable or in some other environment. Naturally, both ends of the link must agree on the FEC method used.

There is OTU and optical data unit (ODU) overhead in the first 14 columns of each row, flowed by two columns of optical payload unit overhead that is used to fit the payload client signals into the OTN frame. The first six bytes of each OTN frame are used for frame alignment, followed by eight bytes of OUT overhead.

The overall structure of the OTN frame is shown in Figure 4.17.

GENERIC FRAMING PROCEDURE

How do OTN frames allow so many different types of client payloads inside the frame? Well, one way is that many potential client signals such as IP packets or

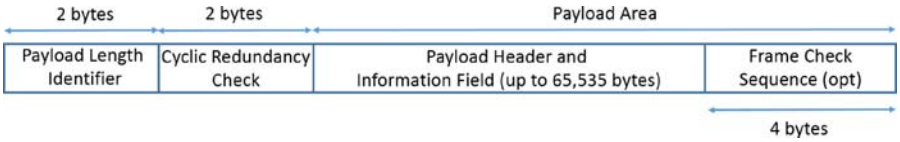

FIGURE 4.18

The GFP frame structure.

Ethernet frames can be adapted to the OTN frame by using a kind of intermediate structure called the Generic Framing Procedure (GFP) frame. IP packets can be carried in OTN by loading them first into ATM cells or SONET/SDH frames, but many IP packets will end up in OTN frames inside Ethernet over GFP frames. Many interfaces use Ethernet already, so it will be easy to adapt these interfaces to OTN using DWDM, but, as mentioned, this is not mandatory. As always, the main requirement is that the equipment on the other end of the span be compatible with whatever method is used by the sender.

GFP frames fall into two main categories: client frames and control frames. Client frames carry the payloads (and some types of payload control information) and control frames carry things like idle frames (to fill the link when no IP packet or Ethernet frame is available) and other link management information.

Unlike OTN frames, the GFP frame has a simple, linear structure. The first two bytes are the payload length identifier (PLI) and the next two bytes are a cyclical redundancy check (CRC-16) to protect the PLI. The payload area that follows can be up to 65,535 bytes long and end with an optional 4-byte frame check sequence (FCS). The payload area includes a payload header that indicates the structure of a payload (Ethernet frame, SONET/SDH, and so on), followed by the client signal information. The general structure of the GFP is shown in Figure 4.18.

FEC RESEARCH AND DEVELOPMENT

Optical software engineers and developers continue to explore more and more powerful varieties of FEC codes. As the minimization of hardware components and the maximization on processing power continues FEC methods that were beyond the capabilities of equipment not so long along will become (like everything else in this field) first possible, then affordable, and finally common. Newer methods might not yet be covered by standards or open implementations, which will impact multivendor interoperability. However, there will always be a market for the first or the best implementation of a given FEC technique, especially if it lowers the overall cost of running the network.

This chapter has explored Hamming codes in depth and introduced BCH and RS code. BCH and RS codes are examples of block codes and cyclical codes, but there are other types of codes that remain an area of active research. Most

notable are the convolutional codes using Viterbi decoders. These codes operate with a sliding sequence of data bits that are used to generate the code words. One form of recursive convolutional code that acts on blocks and can be used in OTNs is called a *staircase code* because of the particular way it operates. This is often referred to as high-gain FEC (HG-FEC).

There are also concatenated codes and turbo codes that are very complex, but offer very high degrees of error correction gain. Many of the codes in use today are known as *hard-decision* FECs. That is, a single parameter such as the amplitude of the incoming signal is examined and a decision is made as to whether this should be a 0 or 1 bit. But when coherent optical devices are used, more than one parameter is used to modulate the digital signal, such as phase and amplitude. But which is more critical to 0 or 1 determination? Is an impulse more critical to the value of the signal than phase? Or is it the other way around? Can this sensitivity vary over time and in different situations such as in submarine cables? FEC codes that take this all into consideration are called *soft-decision* FEC codes and are important in coherent optic implementations.

OTN FOR THE ILLUSTRATED NETWORK

To add FEC to the links on the Illustrated Network, we would have to upgrade the links from SONET/SDH to 10 Gbps. In practice, this could be a long and tedious process, but in a book like this, it's as easy as changing the Juniper Networks SONET/SDH hardware interface port format (e.g., so-0/0/0) to the interface format for 10G (xe-0/0/0). Juniper Networks also supports DWDM OTN 100G interfaces, which have the form et-0/0/0.

We haven't examined the full configuration of interfaces yet, or how to assign IP addresses, so we'll confine ourselves to the FEC methods for 10G and 100G interfaces and use the FEC codes available for each. Juniper Networks line cards can detect and correct up to 1 in every 70 bits in error, on average. This can be up to 1.9 billion error corrections per second, all in real-time and done without resending any information. That kind of error rate without packet loss is what makes FEC so attractive.

If we examine the FEC options for 10G interfaces, we'll find three varieties of FEC listed. Table 4.4 lists their common name in the parameter, the formal name, and the originator of the method.

The ITU has been discussed before (G.975 is a standard for FEC on submarine cables, but the FECs can be used in other types of links, of course), but AMCC stands for Applied Micro Circuits Corporation (also abbreviated APM, as shown in the table). Cortina Systems developed the Ultra-FEC method and was acquired by Inphi afterwards, which is also reflected in the table. It's not unusual for private companies to pioneer new methods that are later adapted by standards bodies: indeed, that's better than standardizing on technologies that have never been implemented (they know who they are).

Table 4.4 FEC Methods for Juniper Networks 10G Interfaces

Common Name of FEC Method	Formal Name	Organization (Originator)
gfec ("RS(255,239)" or "G709 FEC")	ITU G.709 Annex A	ITU G.975
efec ("Enhanced FEC")	ITU G.975.1 Clause I.4	AMCC (APM)
ufec ("Ultra FEC")	ITU G.975.1 Clause I.7	Cortina/Inphi

These three methods are for Juniper Networks' DWDM and OTN hardware, but most other vendors support them, although interoperability is not always given. Because gfec/efec/ufec are so common, they are sometimes called the "tri-FEC" methods.

What 100 G interfaces? At this high data rate, FEC gets interesting. Not too long ago, all FEC methods were hard-decision methods that decoded the inbound signals bit-by-bit and dealt with only simple "is it a 0 or is it a 1?" situations because direct-detect optics were used to recover these individual bits from the optical-to-electrical domain.

But once coherent optical transmission came along with DWDM and OTN, receivers took multi-bit samples of the incoming signals and looked at more than one signal parameter before creating electrical 0 and 1 strings. This led to the development of more powerful and intelligent FEC methods. These newer FECs were called soft-decision FECs to distinguish them from their hard-decision predecessors. Both perform the same basic tasks (error correction and signal gain) but SD-FECs were better at the task (at the cost of more delay and processing power required).

In other words, SD-FEC can correct more errors and tolerate more noise on a link over greater distances than HD-FECs can. Table 4.5 shows the FEC parameters that can be used on Juniper Networks' 100G Ethernet interfaces by common name, formal name, and FEC type.

For wireless networks, soft-decision low-density parity-check (LDPC) FECs are very popular. Soft-decision FEC codes based on Turbo codes (Turbo product codes or TPC), which have very high processing requirements, are also becoming more common. Note that proprietary or vendor-specific implementations have interoperability issues that standard methods are intended to avoid (but that does not mean there are none: it just means no one should say "You should know better. . .").

If performance is more of a concern than compatibility, you should use efec or ufec for 10 Gbps and a soft-decision over a head-decision method for 100 Gbps. In many cases, gfec is the default method and the others must be explicitly configured.

Once we've configured, e.g., sd-fec-ldpc on a 100G link, how do we know that the FEC method is doing its job? All routers and related devices will have ways to examine the operation of the interface, and for Juniper Networks devices on the Illustrated Network, we can use the command show interfaces . . . extensive.

Table 4.5 FEC Methods for Juniper Networks 100G Interfaces

Common Name of FEC Method	Formal Name	FEC Type
gfec ("RS(255,239)" or "G709 FEC")	ITU G.709 Annex A	Hard decision
hg-fec ("High-gain FEC", "staircase")	NA (proprietary)	Hard decision
sd-fec-ldpc	NA (Juniper-specific)	Soft decision
sd-fec-tpc	NA (Juniper-specific)	Soft decision

Table 4.6 FEC Methods for Juniper Networks 100G Interfaces

FEC Performance Parameter	Definition	Notes
Corrected errors	Number of bits received that were in error but corrected	Count of corrected errors (should be viewed along with uncorrected words)
Uncorrected words (UCW)	Number of code words received that could not be corrected	A non-zero UCW count is good and means no packet loss
Corrected Error Ratio	Number of corrected bits divided by number of bits received	An estimate of the BER on the link. A good overall measure of link health.

Here's example FEC performance output:

```
...
OTN FEC statistics:
  Corrected Errors    13032402
  Uncorrected Words   0
  Corrected Error Ratio (  3 sec average)  3.42e-05
...
```

What do these numbers mean? The answer is in Table 4.6.

QUESTIONS FOR READERS

1. What are some differences between hard decision versus soft decision error control?
2. What is the meaning of the (x,y) notation in error correction codes?
3. Using the (7,4) example in the chapter, the string 0110101 is received as 0110100. What data or parity bit is in error and what is the syndrome?
4. How is the syndrome used to correct the bit in error?
5. What is the difference between "in-band" and "out-of-band" FEC in optical networks?