The EvDO Forward Link Transmission Protocol

There are numerous Cellular Communications technologies. What they all have in common is that they are all basically glorified walkie-talkies (2 way radios) and they all engage in the same basic four steps to accomplish what they do. Take a look at Figure 1. The cell tower transmits data intended for the cell phone (step A). The cell phone receives that data (step B). The cell phone transmits data intended for the cell tower (step C). The cell tower receives the data transmitted by the cell phone (step D).

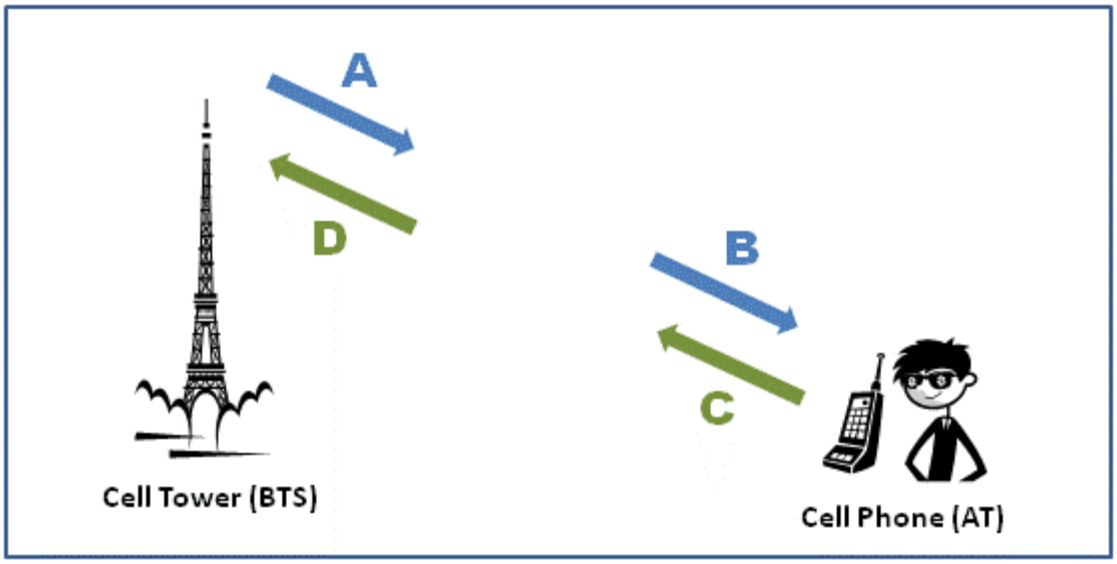

Figure 1

(590,320)

The document explains how step A (transmission from cell tower to cell phone) works for the cellular technology known as EvDO (Evolution – Data Optimized). EvDO itself has gone through some evolution, with new capabilities being added over time. This document covers the basic, original EvDO. Once you understand that, understanding the added enhancements should be a relatively minor effort.

Some important terminology: Steps A and B together constitute the "forward link". Steps C and D make up the "reverse link". The generic term for any device that communicates with a cell tower (cell phone, laptop with broadband wireless, etc) is an "access terminal", or AT for short. The technical term for a "cell tower" is a "BTS". So this document covers the sending of data from a BTS to an AT.

Once you understand step A, understanding step B is relatively simple (reception being mostly just the inverse of transmission). However, be aware that steps C and D (the reverse link) are completely different from steps A and B (the forward link). That is not too surprising when you think about it. After all, the cell tower has essentially unlimited power; the AT has only a small battery. The tower must manage simultaneous communication with multiple AT's, while a given AT only cares about its own communication.

Again, this document only covers the forward link transmission protocol (step A). Let's get started.

Framing the Problem to be Solved

OK, so we know the cell tower is going to transmit data that is intended to be received by an AT. What kind of data will a cell tower actually be sending? Here's the list:

Pilot "Data"

The first thing that has to happen is that the cell tower has to make itself known to the AT (or to look at it from the other direction, the AT has to find the cell tower – think about that "searching for network" message you see when you first turn on your cell phone). Basically, the tower transmits a "beacon" (think of a light house for purposes of analogy) that allows the AT to determine that a cell tower is present. In terms of digital content, this beacon is nothing but a never ending stream of 0's (hence the reason putting the word "data" in quotes in the title). This beacon is referred to as the "***Pilot Channel***". Besides alerting the AT to the presence of the cell tower, the AT can also use the pilot to get a rough idea of how far away the tower is. IE, is the pilot signal strong (tower is near) or weak (tower is far).

Sharing Data

There is only a single "ether" out there. Everybody shares the same airways. If this radio communication business is actually going to work, there has to be some way for the users to coordinate so that they don't stomp all over each other. There are lots of ways this is done – for example by government regulation: assigning different frequencies to different purposes. In the case of a single cell tower, it has to coordinate how multiple ATs use the assigned frequencies in its immediate location. To achieve this goal the tower is constantly sending out a stream of information about how the ATs should transmit their data. All the ATs listen to this information, and adjust their actions accordingly. This stream of information is called the "***MAC Channel***". (MAC stands for Media Access Control).

Control Data

In addition to the split-second by split-second updates that are sent on the MAC Channel, the cell tower sometimes has less frequent control information to send out. For example, if someone is trying to call your cell phone, the cell tower has to send a notice (called a "page") to your cell phone letting it know that someone is trying to call. This less frequent control information is sent on what is called the "***Control Channel***".

User Data

This is your digitized voice, the web page you want to download, the text message you just received. This is the reason the whole cellular infrastructure was built – the useful information you want to get to or from your AT. User data is sent over what are called "***Traffic Channels***". A cell tower will have many traffic channels. Some of these traffic channels may be of the "broadcast" flavor – listened to by multiple ATs (think many people watching a ballgame on their smart phones), and some will be dedicated to individual users (think about you reading your email on your smart phone).

A Few Words About Channels, Bits, Chips, and Symbols

Unfortunately, in the EvDO world everything is called a channel:

> *This channel is based on the first channel, which is placed in a channel composed of the previous channel to correlate with the next channel, unless the channel has failed to initialize the channel for a channel in the channel.*

The TV show South Park actually did an episode about this – check out

http://www.southparkstudios.com/clips/151555/marklar-to-marklar

I will try to do better.

The 4 channel types mentioned above (Pilot, MAC, Control, Traffic) are "***logical channels***". They represent a well defined stream of data that needs to be sent – but the term "channel" does not imply in any way that the mechanism by which the data is sent is the same. For example, do not assume that the way the MAC channel information gets sent out is the same as the way the Traffic channel data gets sent out (in fact they are quite different).

Logical channels are either "***broadcast***" (intended for multiple recipients) or "***non-broadcast***" (intended for a single recipient. The Pilot, MAC, and Control channels are inherently broadcast – in fact, all AT's are expected to listen to all 3 channels all the time. Traffic channels may be either broadcast or non-broadcast. However, in the case of a broadcast traffic channel it is not the case that all AT's will listen to it (EG, not all AT owners will be interested in watching that ball game – or for that matter, not all of them will have paid to be able to watch the game on their smart phones).

Also, the spec for EvDO (IS-856) shifts between calling things bits and chips and symbols, seemingly at random. This is somewhat understandable, as one step's chips may become another step's bits. But still, it is very confusing. Again, I'll try to be clearer.

Limitations

The data in this document is certainly at least *mostly* accurate – but I can't guarantee that every detail is absolutely correct. Also, this document describes Rev-0. Rev-A is *mostly* the same, but does add some features and complications.

The Next Sections

The next few sections will cover a series of distinct topics that pertain to understanding the EvDO forward traffic channel. Once the necessary background is in place we can put the pieces together into a coherent and complete picture.

Time Division Multiplexing (TDM)

The primary way that the EvDO forward link is shared among multiple AT's is by Time Division Multiplexing (TDM). Time is broken up into "slots", where each slot lasts 1 2/3 (1.67) milliseconds. Each slot is used to send data to a particular user (non-broadcast) or to send data for a broadcast channel. An example of slot usage might be:

Slot 1 – send data to Ann
Slot 2 – send data to Bill
Slot 3 –send data to Charlie
Slot 4 – send data to Diane
Slot 5 – send data to Bill
Slot 6 – send ball game data
And so on …

Slots are also used to send Control Channel information.

So – at the start of each slot the cell tower has to decide what data to send during that slot. Should it send control data? Or should it send user data? If user data, which user's data should get sent? Or should it send broadcast data? These questions are rhetorical – the component in the cell tower that decides what data should be sent is called the scheduler. This document is not concerned with how the scheduler makes that decision. Rather once the decision has been made, we are learning how that particular data get sent out over the air.

Slots are also combined to make larger units, and broken down into smaller units as well. Look at Figure 2. Sixteen slots make a "***frame***", and 16 frames make a "***control channel cycle***". Finally, 12 control channel cycles make a "**sleep cycle**". Going the other way, a single slot is broken up into two "***sub-slots***", and each sub-slot is broken up into 5 "pieces" (pieces is not a technical term, but I had to call them something).

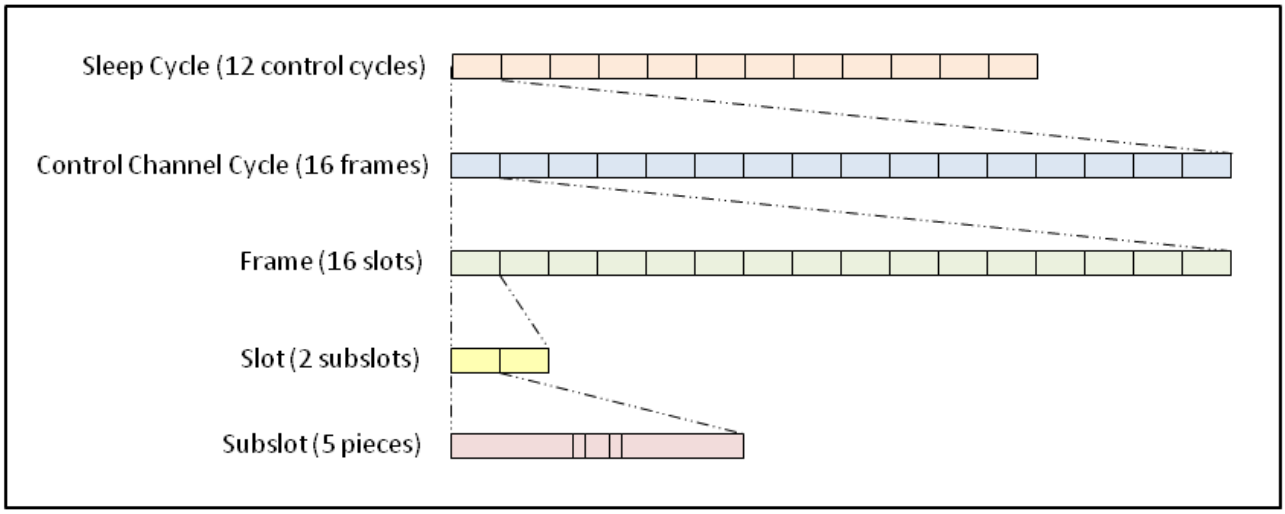

Figure 2 (870,370)

You'll notice I talked about a slot being used to send either Control Channel data or Traffic Channel data. What about Pilot Channel data and MAC Channel data? They do not get dedicated slots. Rather, a small

amount of *every* slot is taken to carry that data (think of it as a tax!). If you look at the subslot in Figure 2, the little center piece is used to carry the Pilot Channel data, and the 2 little pieces on either side of that are used to carry the MAC Channel data. This reflects the facts that 1) we want the AT to be able to find and lock onto the pilot quickly, and 2) the MAC data is changing very rapidly. By sending the pilot data twice every slot, and the MAC data 4 times every slot, the ATs get very quick access to that data. So, for example, when a scheduler decides that a given slot is going to be used to send Barry his data, in fact Barry only gets to use 78.125% of the slot. 9.375% is "stolen" for the pilot data, and another 12.500% is "stolen" for the MAC data.

So – we now know that we'll be slicing time up into 1.667 millisecond slots. The question arises: Can we start doing our slicing at any moment we like? The answer is no. It is a requirement that the start of a frame begin on a 2 second boundary, where the time source is taken from a GPS signal. So you can start slicing at (for example) 10:15:00, or 10:15:02, or 10:15:04. But not at (for example) 10:15:01.425. Why an even second boundary? There are 600 slots in a second, and 16 slots in a frame. 600 does not evenly divide 16 (it gives 37.5). However, 2 seconds (1200 slots) gives us exactly 75 frames – so once we start a frame on an even second boundary, all other even second boundaries will also line up with frame starts.

Time synchronization is critical in EvDO because cell phones may be in contact with multiple cell towers. If each cell tower just transmitted its slots "whenever", it would make the cell phones job much harder. With all transmissions from all cell towers being in sync, it is easy for the cell phone to track them all (and, for example, to switch from using one cell tower to another).

To summarize – the primary sharing mechanism for EvDO Forward Transmission is Time Division Multiplexing. The key timing element is the "slot". It is on a per slot basis that decisions are made about what data to send (in point of fact, other key decisions are also made on a per slot basis, but we'll get to that later).

Code Division Multiplexing (CDM)

Another method of sharing a communication link is Code Division Multiplexing (CDM), and EvDO uses this as well. The usual informal analogy used to explain CDM goes as follows: There is a large cocktail party, and everyone is speaking at once. None the less, the people are able to hold intelligible conversations because each little group is speaking a different language. Due to the miracle of the human brain, the group speaking French is able to tune out the German conversation, the Italian conversation, etc, and pay attention only to their French colleagues.

Similarly, in a CDM system each user gets a unique "code", and by listening only to that code he hears only his own data, even though data is being sent at the same time to multiple other users (who each have their own code).

There are two types of codes in use in EvDO:

1) Walsh Codes

These are fixed length codes (some power of 2) that are mathematically "orthogonal", meaning that for CDM purposes they are "perfect", in the sense that the coding and decoding of multiple streams of information should work with maximum effectiveness. For a given length N, there will be N available codes. For example, with a length of 64 bits there are 64 distinct Walsh codes, meaning that up to 64 different streams of data can be multiplexed together. For a code length of 16 bits there would be 16 Walsh codes, and so on. The length of a Walsh code is indicated by a subscript. IE $Walsh_{16}$, $Walsh_{32}$, $Walsh_{64}$, etc.

2) PN Codes

These are **P**seudo-random **N**oise bit sequences. They appear random but are in fact completely deterministic. For CDM coding purposes they are not perfect in the sense that Walsh codes are, but they are "good enough" so that a receiver can still decode his unique data in the presence of data transmitted with other PN codes. PN codes are much longer than the Walsh codes. The EvDO "Long PN Code" is 32,768 bits long (contrast that with typical Walsh code lengths of 16, 32, or 64 bits). Not coincidentally, 32,768 bits corresponds to 16 slots – IE, one frame. So one frame is the amount of time it takes to go through the EvDO long PN code once.

In actuality, the EvDO use of CDM is quite complex and a bit tricky. The "each user gets a unique code" idea is actually used twice - once on the per-user level for MAC data, and once on a per-cell-tower level. In addition, EvDO uses CDM to convert (in a sense) a serial radio link into a parallel radio link. Let's take these one at a time.

1) Per user code for MAC data

Each active user gets a unique 64 bit Walsh code that is used to identify MAC data intended for him. One 64 bit code is reserved for broadcast MAC data (MAC data that everyone should be listening to).

2) Per cell-tower code

In EvDO, adjacent cell towers may be broadcasting on the same frequency. To avoid interference, each cell tower gets a unique PN code. The AT, by decoding based on a particular PN code, is selecting the data being sent by the matching cell tower, and is ignoring data sent by other cell towers using different PN codes.

3) Parallelization codes

This is the tricky one. Normally, CDM is used to transmit data from different users at the same time. However, there is nothing to prevent one from transmitting data from the same user at the same time.

Let's take an example. Suppose we have two codes (A and B), and a time period in which we will transmit 512 bits. The normal usage would be to give user 1 code A and user 2 code B. Both user 1 and user 2 would get to transmit 512 bits in the time period. But we could give both codes to user 1. User 1 will then be able to transmit 1024 bits in the same time period (512 bits using code A and 512 bits using code B). User 2 is out of the picture in this scenario. In essence, user 1 was able to split his 1024 bits into two 512 bit streams and send them in parallel. EvDO in fact uses this technique.

Note that most of the data being sent out will in fact be doubly CDM encoded. The MAC data will be CDM encoded with a 64 bit Walsh code, and then THAT output will be CDM encoded by the cell tower PN code. The control and user traffic will be CDM encoded with the parallelization codes, and then THAT output will be encoded by the cell tower PN code.

It should also be pointed out that the CDM codes are not a security mechanism. The codes are all public knowledge. This is not to say that cellular communication is not secure – but security is provided by other means, not by the use of CDM codes. The CDM codes are strictly to facilitate sharing of a common resource (radio bandwidth).

Error Detection and Correction, and Data Conditioning

Radio communication is inherently error prone – the airwaves are subject to all sorts of uncontrolled phenomena. Hence error detection and correction is critical, and EvDO uses "turbo codes" for this purpose. There are many varieties of turbo code, and the EvDO forward link uses two of them. The gory details are in the spec, but for our purposes all we need to know are that one expands the number of bits to be sent by a factor of 3, and the other expands the bit count by a factor of 5.

For example, if you start with 1024 bits of data, after turbo encoding is done you will have either 3072 or 5120 bits to be sent. The 1 to 5 expansion gives you better error correction, but is more expensive (obviously) than the 1 to 3 expansion. Which one is used is determined on a packet by packet basis, based on the desired transmission speed and a guesstimate of the air link conditions. Note that turbo-encoding introduces a not insignificant amount of overhead in the transmission of the data.

In addition to the turbo encoding, EvDO also massages the bit stream is various other ways. The exact reason for these manipulations is mostly left to the imagination of the reader of the spec, but we can at least call them out:

1) Scrambling

The turbo encoded data is XOR'ed (exclusive or'ed) with a dynamic but well defined bit stream. The receiver of the data will recover the original turbo encoded bits by again XOR'ing the received data with the same dynamic but well defined bit stream. That this works is a result of the logical identity:

((X * Y) * Y) == X (where '*' stands for the exclusive or operation)

According to the spec, the scrambling will "randomize" the data.

2) Reordering and Permutation

The scrambled bit stream is further massaged by taking the bits in groups of either 3 (if 1/3 turbo-encoding was used) or 5 (if 1/5 turbo-encoding was used) and shuffling them around. The actual shuffling algorithm is quite complex, and there seems to be no particular value to describing it here (see the spec if you want to try and figure it all out). However, the action is deterministic, so the receiver can do the same thing in reverse to recover the original bit stream.

Note that scrambling and reordering and permutation do NOT increase the number of bits that need to be sent. All the "bloating" is done in the turbo-encoding step.

Physical Packet Size and Format

Only a fixed set of packet sizes are allowed at the physical layer. The allowed sizes are: 256, 512, 1024, 2048, 3072, and 4096 bits. Furthermore, only control data packets can be 256 or 512 bits long (IE, user data is not sent in packets smaller than 1024 bits). If the amount of data to be sent is less than the maximum a packet can handle the remainder of the packet is filled with zeroes. So, for example, if only a single byte is to be sent to a given user, it will take an entire 1024 bit packet to send that byte (the 256 and 512 bit lengths are not allowed for user traffic, and the EvDO framework is smart enough not to use a 2048, 3072, or 4096 bit packet when a 1024 bit packet will do).

There are four types of subfields that appear in a physical layer packet:

1) Payload

This is the actual user data (from an upper level in the protocol stack).

2) FCS

Every physical layer packet has a 16 bit check sum, which allows the receiver to validate the contents.

3) Tail

Every physical layer packet has a 6 bit "tail" field that provides info about the packet.

4) Pad

Certain packet sizes require padding to keep the desired alignment. Note that this different from the payload padding that is done when the user does not provide enough bits to completely fill the payload section. So in a given packet, padding may occur twice. First, on the payload bits, to completely fill out the payload field. Second, between the Payload/FSC/Tail fields to have them align properly. This is covered in more detail below.

Figure 3 gives the layout of each of the physical packet sizes. The field sizes are not strictly to scale. Note that the amount of payload space varies from 232 bits (for a 256 bit packet) up to 4008 bits (for a 4096 bit packet). Also note that in the larger size packets the payload bits are not contiguous.

I want to mention here that the relationship between physical packets and TDM slots is variable. A single physical packet will take anywhere from 1 slot up to 16 slots to transmit, depending on the size of the packet and the data transmission rate selected. This is covered in more detail later.

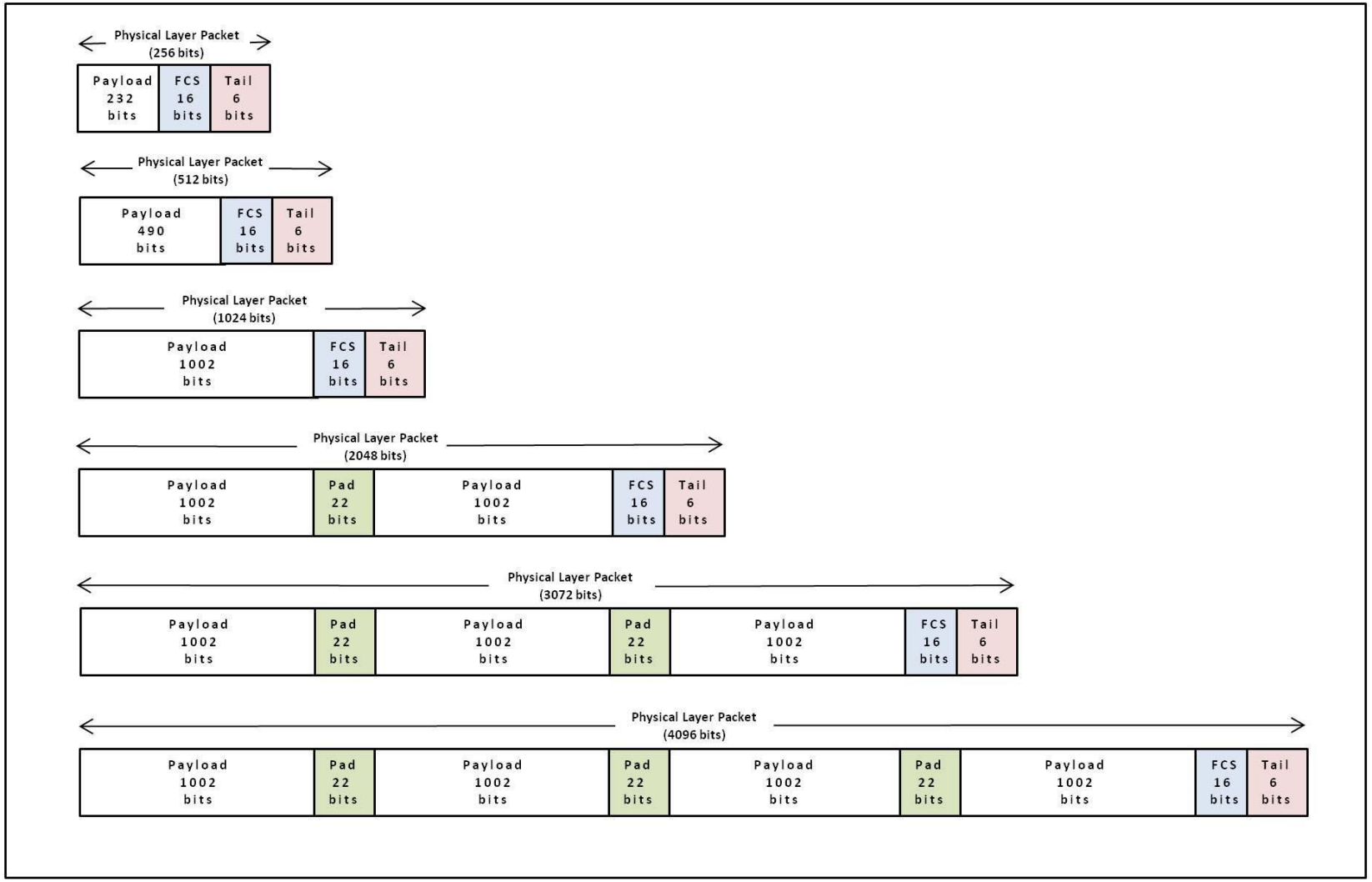

Figure 3

(960,640)

Bits, Chips, and Symbols

At this point, some discussion of bits, chips, and symbols is in order.

Hopefully everyone is familiar with the concept of a 'bit' – a minimal two state piece of information – the two states usually being represented as '0' and '1'.

When CDMA spreading is done on some data each bit gets turned into some (fixed) number of 'chips'. For example, in the EvDO forward link, each bit of MAC data gets expanded into 64 chips. Now each chip is also either a '0' or '1', so in a sense chips are also bits. To make sense of this you have to think in terms of layers. EG, bits at layer 1 get turned into chips to be given to layer 2. However, chips are still just zeroes and ones, and layer two doesn't know that the zeroes and ones he's getting are chips – to him they are just bits, and he just does normal binary data processing on them.

Symbols are pieces of information that can have some fixed number of states (usually a power of two). In terms of their actual implementation – symbols may be represented using bits. For example, with 3 bits we can represent a set of 8 symbols.

All of this is by way of warning – in trying to follow what's going on in the EvDO forward link we are constantly shifting between bits, chips, and symbols – and vigilance is required to keep straight what is going on.

Figure 4 illustrates the ideas. We start with three bits. Each bit is spread by a factor of eight, giving a total of 24 chips. The chips are then taken in groups of 3 to create eight (8-level) symbols. We can represent the symbols using eight distinct bit patterns. We can also represent the symbols using 8 distinct letters (eg A thru H). We could also, if we wanted to, represent the symbols using eight distinct pictographs: ✈ ☼ ♦ ❄ ✞ ✞ ✙ ✠ ✡ ☪ The key point is to keep the idea of a 'symbol' distinct from its physical representation.

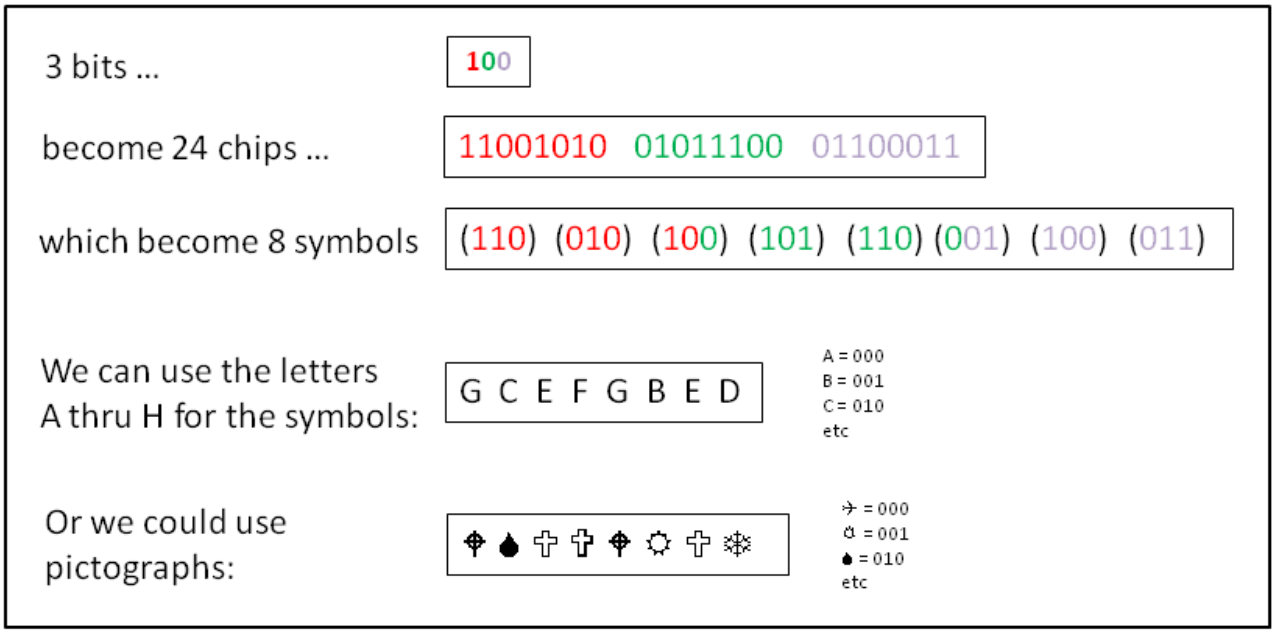

Figure 4

(850,450)

Variable Transmission Rate

A key feature of the EvDO Forward Transition of data is that it is not a fixed data rate – the rate can vary from 38.4 kbps to 2457.6 kbps. It is actually the AT who requests a specific data transfer rate. Once a data rate and a physical packet size have been selected, a bunch of other relevant parameters are then fixed. This is shown in the following table. Note that not all combinations of data rate and packet size are supported.

Data Rate (kbps)	**Packet Size (bits)**			
	1024	2048	3072	4096
38.4	PrefixRep = 32 PacktRep = 9.6 SlotCount = 16 CodeRate = 1/5 ModType = QPSK			
76.8	PrefixRep = 8 PacktRep = 4.8 SlotCount = 8 CodeRate = 1/5 ModType = QPSK			
153.6	PrefixRep = 8 PacktRep = 2.4 SlotCount = 4 CodeRate = 1/5 ModType = QPSK			
307.2	PrefixRep = 4 PacktRep = 1.2 SlotCount = 2 CodeRate = 1/5 ModType = QPSK	PrefixRep = 4 PacktRep = 2.04 SlotCount = 4 CodeRate = 1/3 ModType = QPSK		
614.4	PrefixRep = 2 PacktRep = 1.0 SlotCount = 1 CodeRate = 1/3 ModType = QPSK	PrefixRep = 2 PacktRep = 1.02 SlotCount = 2 CodeRate = 1/3 ModType = QPSK		
921.8			PrefixRep = 2 PacktRep = 1.02 SlotCount = 2 CodeRate = 1/3 ModType = 8-PSK	
1228.8		PrefixRep = 2 PacktRep = 1.0 SlotCount = 1 CodeRate = 1/3 ModType = QPSK		PrefixRep = 2 PacktRep = 1.02 SlotCount = 2 CodeRate = 1/3 ModType = 16-QAM

1843.2			PrefixRep = 2 PacktRep = 1.0 SlotCount = 2 CodeRate = 1/3 ModType = 8-PSK	
2457.6				PrefixRep = 2 PacktRep = 1.0 SlotCount = 1 CodeRate = 1/3 ModType = 16-QAM

Table 1

PrefixRep – How many times the 32 symbol prefix is repeated
PacketRep – How many times the packet data is retransmitted. Fractional values are allowed because one may need to pad out to a slot boundary.
Slot Count – the number of slots it will take to send the packet (including retransmissions)
CodeRate – how much the turbo encoding "bloats" the bit count – either a factor of 5 or a factor of 3
ModType – how to modulate the signal: QPSK = 2 bits per symbol; 8-PSK = 3 bits per symbol; 16-QAM = 4 bits per symbol

Note that info for 256 and 512 bit packets is missing from the table. It also seems to be mostly missing from the spec. As noted, only control data can use the 256 & 512 bit sizes. Also, the spec does say that only the 38.4 and 76.8 speeds are allowed for those packet sizes.

First Pass at a Big Picture

Figure 5 shows the EvDO Forward Transmission Layer as a block box.

On the input side we have all sorts of packets that 'want' to get transmitted. Conceptually, the input interface to the physical layer is:

```
SendPacket ( ptr, length,  user_id,  rate );
```

Where ptr is a pointer to the data to be sent, length is the number of bits (256, 512, 1024, etc), user_id indicates who (which AT) should get this data, and rate is the data rate to be used for the transmission.

On the output side we have the interface to the radio that actually broadcasts the signal. It's a digital interface, defined in terms of symbols. The physical layer is responsible for giving symbols to the radio at a fixed rate of 1,228,800 symbols per second, always – regardless of data rate, packet size, or anything else.

The job of the physical layer is to take those packets, with their variable lengths and various desired data rates, and convert them into the constant stream of symbols that the radio wants.

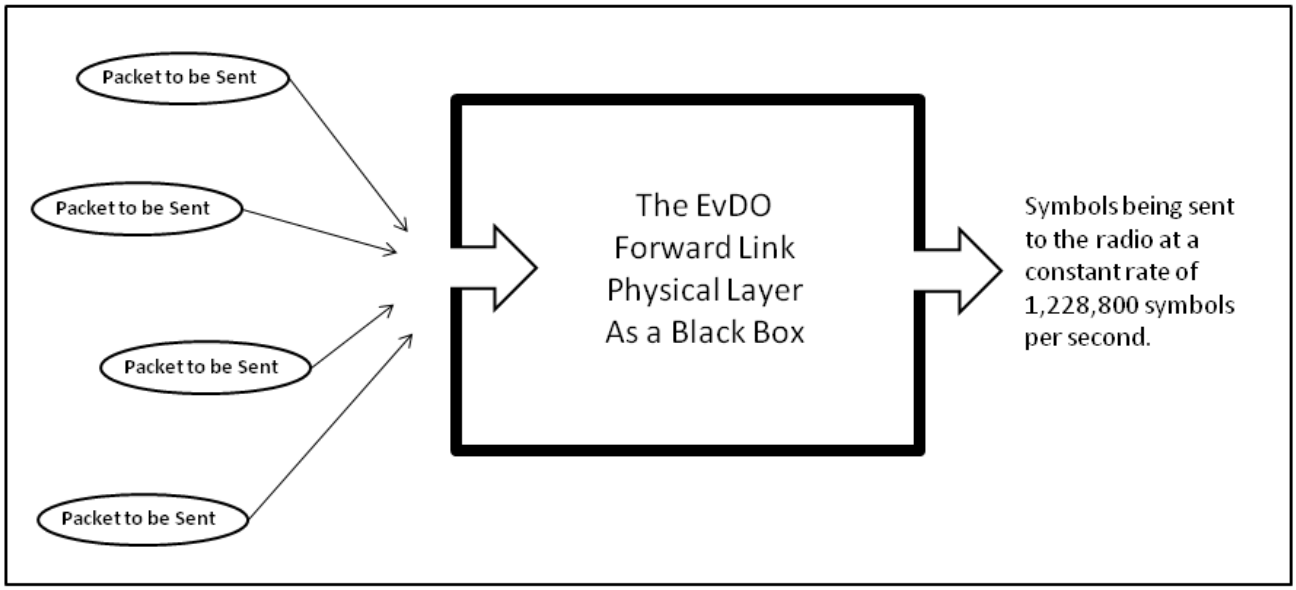

Figure 5 (910,460)

IQ Data, Constellations, and Radio Symbols

As previously mentioned, the EvDO physical layer is required to send a constant stream of symbols to the radio. So, just what exactly does one of these symbols look like? Each symbol is a pair of signed 6 bit numbers. Here's an example of one symbol (I'm expressing the numbers in decimal):

(28,-17)

We have to send 1,228,800 such pairs every second. For physical/historical reasons, these pairs are referred to as IQ data (the first number is I, the second number is Q). This is analogous to the usual convention for plotting data in graphs where you have pairs of numbers called (X,Y).

Just what does such a symbol represent? Well, it will represent either 2, or 3, or 4 bits – depending on what data rate was agreed to between the AT and the BTS. Without knowing what that agreement is you can't really know what the symbol means. How do we go from the symbol (pair of numbers) to the correct bit pattern? We use what is called a "constellation". There is a different constellation for each of the three mapping choices (IE the 2,3, or 4 bit mapping). Figure 6 shows the three constellations:

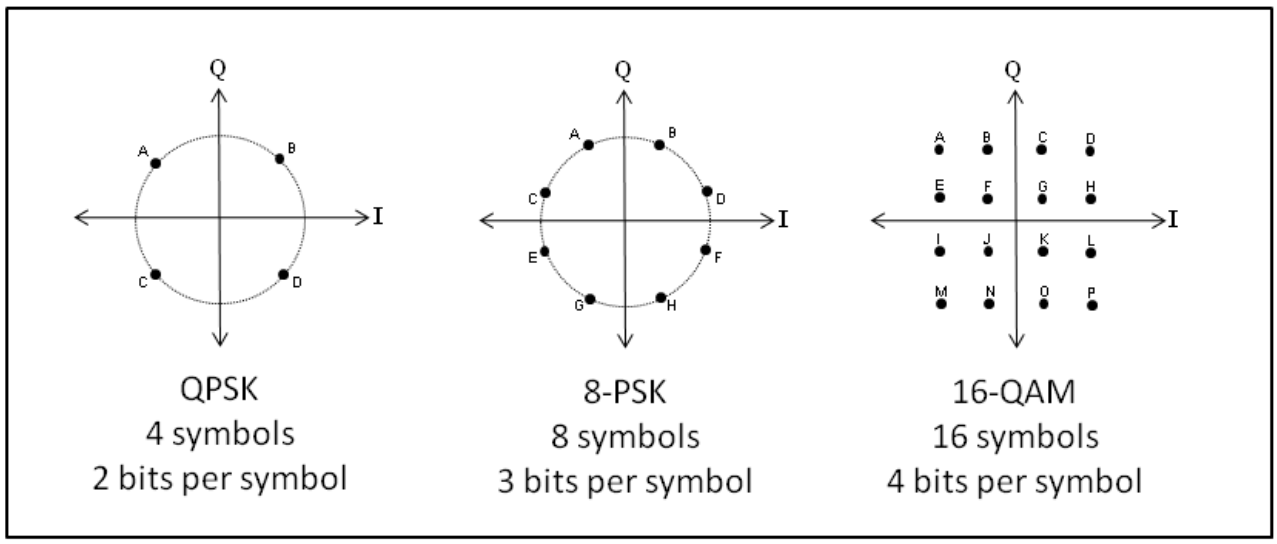

Figure 6

(880,400)

Let's walk through an example. Suppose we've done a bunch of processing on our packet, and now we have a string of 0's and 1's (they may be bits or chips) that is to be translated into symbols. Let's suppose we're going to use QPSK. With QPSK there are 4 symbols (A,B,C,D), each representing 2 bits (or 2 chips). What we do is we grab the bits 2 at a time and map them into A/B/C/D using the following key:

A = 01
B = 00
C = 11
D = 10

We then locate the corresponding spot in the QPSK constellation, and get it's I and Q coordinates, and that pair becomes the 'symbol' we pass to the radio. The exact value will depend on the scale used in I and Q axis, but we can see that for the symbol A, the I value is negative and the Q value is positive. So, what gets passed to the radio would be something like (-40,40)

All of this may seem very convoluted (and it is). The reason behind it may be a bit clearer if we look at what happens on the receiving end as well. To recap – we've taken 2 bits, and converted then into two 6 bit signed integers, which we pass to the radio. The radio does a digital to analog conversion, and transmits a signal through the air.

On the receiving end the radio takes the analog signal and converts it to digital, and passes the converted values as symbols (pairs of 6 bit signed integers) to the receiving software, whose job it is to change the symbol back into bits. Take a look at figure 7, where we've marked out both the transmitted symbol and the received symbol with a red circle:

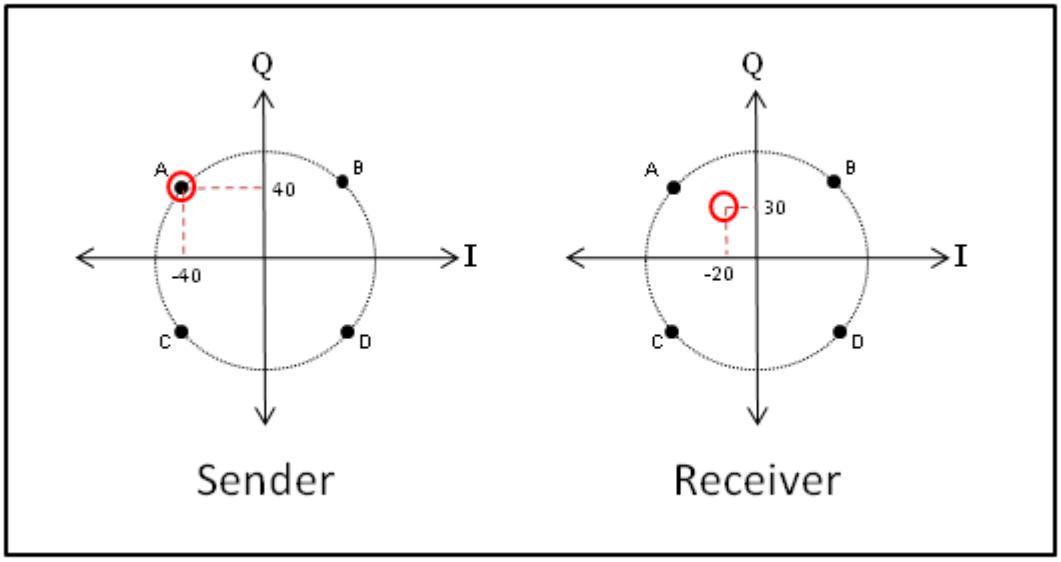

Figure 7

(590,340)

So what went into the transmitting radio was the IQ pair (-40,40), and what came out the receiving radio was the IQ pair (-20,30). What happened? Reality! Radio signals are subject to sorts of problems – interference, fading, multipath, whatever. What you get out ain't necessarily exactly what you put in.

So the receiver received the symbol (-20,30), and it doesn't map to any valid point in the constellation. What to do? Even though there's no exact match, it's obvious that the best match is 'A'. So that's what the receiver does – it takes each IQ pair, and assigns to it the closest matching constellation point.

And that gives us the answer to why we do things this way. We take 2 bits, and convert it to a symbol consisting of two 6 bit integers (a total of 12 bits) – a bloat factor of 6. The benefit from that bloat is additional robustness – the receiver can correctly decipher the intended symbol even under noisy conditions.

CDMA Spreading of Data

As mentioned earlier, EvDO uses CDMA to "parallelize" a single user's data. This section describes that operation in detail. This process happens after the user data has been converted to symbols (IE, the packet has been turbo encoded, scrambled, and divided up into symbols). Look at Figure 8:

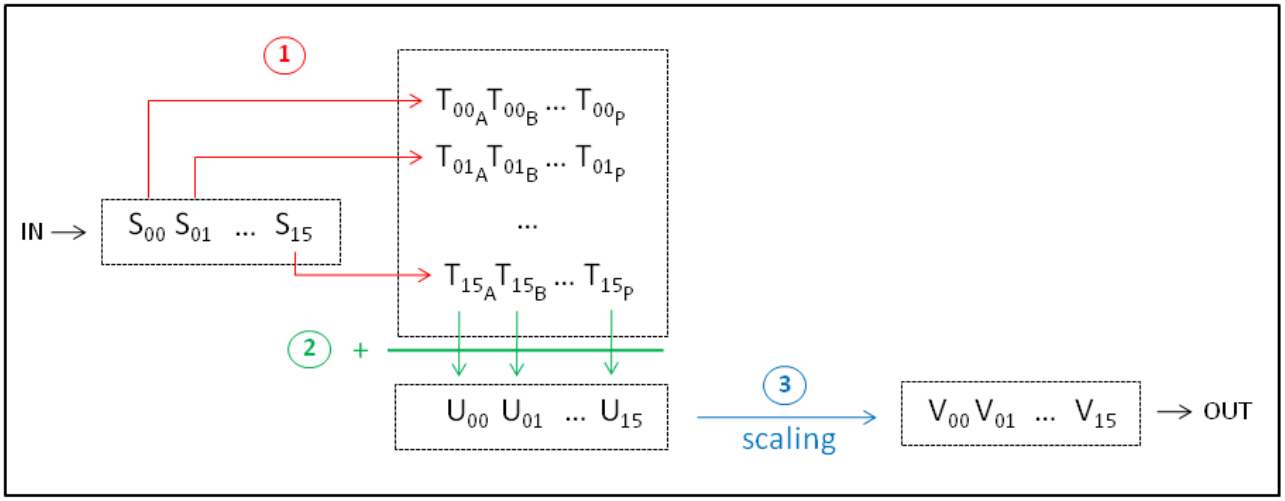

Figure 8 (940,400)

The symbols are handled in groups of 16. In the figure, S_{00} through S_{15} represent one such group of 16 input symbols. Each of these symbols is spread using a different 16 bit Walsh code, producing a total of 256 new intermediate symbols (T_{00a} through T_{15p} in the figure). Thus, for example, from input symbol S_{00} we get intermediate symbols T_{00a} to T_{00p}. This is step 1 in the figure.

Next, the columns of intermediate symbols are added, producing a set of 16 second-intermediate symbols U_{00} through U_{15}. This is step 2 in the figure.

Since we've added symbols together, our value range for the sums is "out of whack". Thus in step 3 the sums are scaled down to the desired range, giving us 16 output symbols V_{00} through V_{15}. Think of it this way, if we add together 16 six bit values, the final result may well be outside what can be represented in six bits. Hence the need for the scaling step.

Note that there is no bloat in this process. Sixteen symbols come in, sixteen symbols go out. Presumably the result of all these machinations is that the final data stream that gets transmitted will be more robust.

Data Repetition

The data to be sent is repeated for 2 reasons:

1) To fill out a slot.

2) To send multiple copies of the data for robustness.

The first reason is trivial. A slot holds a fixed number of symbols. If after converting a packet to symbols the total number of symbols is not an even multiple of the number of symbols per slot, then we end up with a "fractional slot". Since there really is no such thing as a "fractional slot", the left over portion of that final slot has to be filled with something, so we just repeat as many of the data symbols as are needed to fill it up.

The second reason is somewhat involved. Under poor signal conditions we just assume that the data is going to need to be sent multiple times to get one good copy to the target. This is mainly at the lower data rates (indeed, the multiple transmissions is one of the reasons the data rate is low!). The following table lists the data rates that use multiple transmissions:

Data Rate	Repetition Factor	Total Number of Slots Needed
38.4	9	16
76.8	4	8
153.6	2	4
307.2	2	4

Now when multiple slots are needed to send the data the protocol could have just sent them one right after the other (EG **1**,**2**,**3**,**4**), but in fact it is much more clever than that. The slots containing the packet data are interleaved by a factor of four, so what gets sent is, for example: **1**,X,X,X,**2**,X,X,X,**3**,X,X,X,**4** where X represents slots being used to transmit some other packet.

Why do this? Well, we know we'll be sending repetitions of the packet data. What happens if the target is actually able to successfully receive the first copy? Sending the additional copies is a waste. By spreading out the slots used it gives the target time to tell the cell tower that it has in fact received the data, and the cell tower can cancel the additional retransmissions.

So, for example, at 38.4 kbps, the number of slots listed (16) is actually a worst case scenario. If signal conditions are actually better than what was expected and the target successfully receives one of the early copies, the total number of slots used will be less (and so the effective data rate will be higher than 38.4 kbps).

Note that the interleave factor is always four, regardless of the number of slots expected to be used (4, 8, or 16).

Packet Prefix

Each packet that gets transmitted is preceded by a prefix. The size of the prefix varies from 64 to 1024 symbols, depending on data rate. The prefix is generated by taking a '0', spreading it out to 32 symbols using a 32 bit code, and then repeating the 32 symbols between 2 and 32 times to get the total number of symbols required.

The 32 bit code used is connection dependent. That is, each active connection will have a unique code. This is not a Walsh code itself – it something called a bi-orthogonal sequence. Since there are up to (about) 64 active connections what we do is take the 32 thirty-two bit Walsh codes, AND their bit-by-bit complements, to give us a total of sixty-four 32 bit bi-orthogonal sequences.

Note that the prefix is a per-packet affair, NOT a per-slot affair. So in the case where a packet takes more than one slot, the prefix will show up only in the first slot.

The purpose of the preamble is to help the receiving AT synchronize with the transmission. Since the data content of the preamble is a known quantity, the AT can verify that the decoding of the incoming signal is working properly.

The following table sums up the size of the preamble for the different data rates.

Data Rate (kbps)	Preamble Repetition Count	Total # of Preamble Symbols
38.4	32	1024
76.8	16	512
153.6	8	256
307.2	4	128
614.4	2	64
921.6	2	64
1228.8	2	64
1843.2	2	64
2457.6	2	64

Example

Let's go through an example, step by step, of converting an 800 bit packet into symbols to be transmitted at a data rate of 153.6 kbps. Note that the size starts out in bits, and then later changes to symbols. Note that steps 1, 2, 3, 7, & 8 bloat the amount of data to be sent, and steps 4, 5, & 6 do not.

Step	Action	Size
0	Start with an 800 bit packet to be sent	800 bits
1	Pad out to 1002 bits	1002 bits
2	Add 16 bit FCS and 6 bit trailer	1024 bits
3	Turbo encode at a 1 to 5 ratio	5120 bits
4	Scramble and interleave the data	5120 bits
5	Divide the bits into symbols (using QPSK, which is 2 bits per symbol)	2560 symbols
6	CDMA multiplex the symbols	2560 symbols
7	Duplicate the data 2.4 times	6144 symbols
8	Prefix 256 preamble symbols to the data	6400 symbols

As will be explained later, each slot has room for 1600 data symbols, so our 6400 symbols will require 6400/1600 = 4 slots, which is exactly right, based on the info in Table 1. Amazing!

MAC Data

We've now covered in some detail the processing of user data and control data. Now we'll discuss the MAC data. There are 3 types of MAC data that the BTS sends to the ATs:

1) The RAB bit

The RAB (Reverse Access Bit) is a single bit of information that is sent in every slot. Every AT extracts the RAB bit (it will be the same bit) from every slot and acts based on that bit. Basically, the bit tells the AT whether or not the BTS is getting overloaded with received data. A '0' means everything is fine. A '1' means the BTS is overloaded and the ATs need to reduce the amount of data they're sending.

2) The RPC bits

This is also a single bit of information sent in every slot, but it is on a per AT basis. EG, if there are 40 open connections, then each slot will have 40 RPC bits (a separate one for each connection). Each AT only pays attention to its own bit. RPC stands for "Reverse Power Control". A '0' means the AT should reduce its transmission strength, a '1' means the AT should increase its transmission strength.

3) The DRCLock bits

Again, this is also a single bit of information sent in every slot, also on a per AT basis. Each AT only pays attention to its own bit. DRC stands for "Data Rate Control". A '0' means the AT is allowed to request a different data rate. A '1' means the AT is locked into its current data rate (whatever that might be).

A key point is that in any given slot, the BTS always sends the RAB bit, but it will send only one of the RPC or DRCLock bit sets. IE, in any given slot the data will consist of either:

RAB + RPC

OR

RAB + DRCLock

For example, suppose there are 50 open connections. Then in each slot there will be 51 (1 + 50) MAC data bits, *NOT* 101 (1 + 50 + 50) data bits.

How is it decided on which set of data to send for a given slot? It is done on a fixed cycle. Every 1 out of N slots will have DRCLock data, the rest of the slots will have RPC data. The value of N is configurable. If, for example, it was 8, then the MAC content of a sequence of slots would look like:

D,R,R,R,R,R,R,R,D,R,R,R,R,R,R,R,D,R,R,R,R,R,R,R etc

Where **D** stands for a slot with RAB+DRCLock data and **R** stands for a slot with RAB+RPC data.

So, for a given slot we will have N+1 bits of MAC data to send, where N is the number of active connections. How do we convert those bits into radio symbols? The process for RPC data explained in Figure 9:

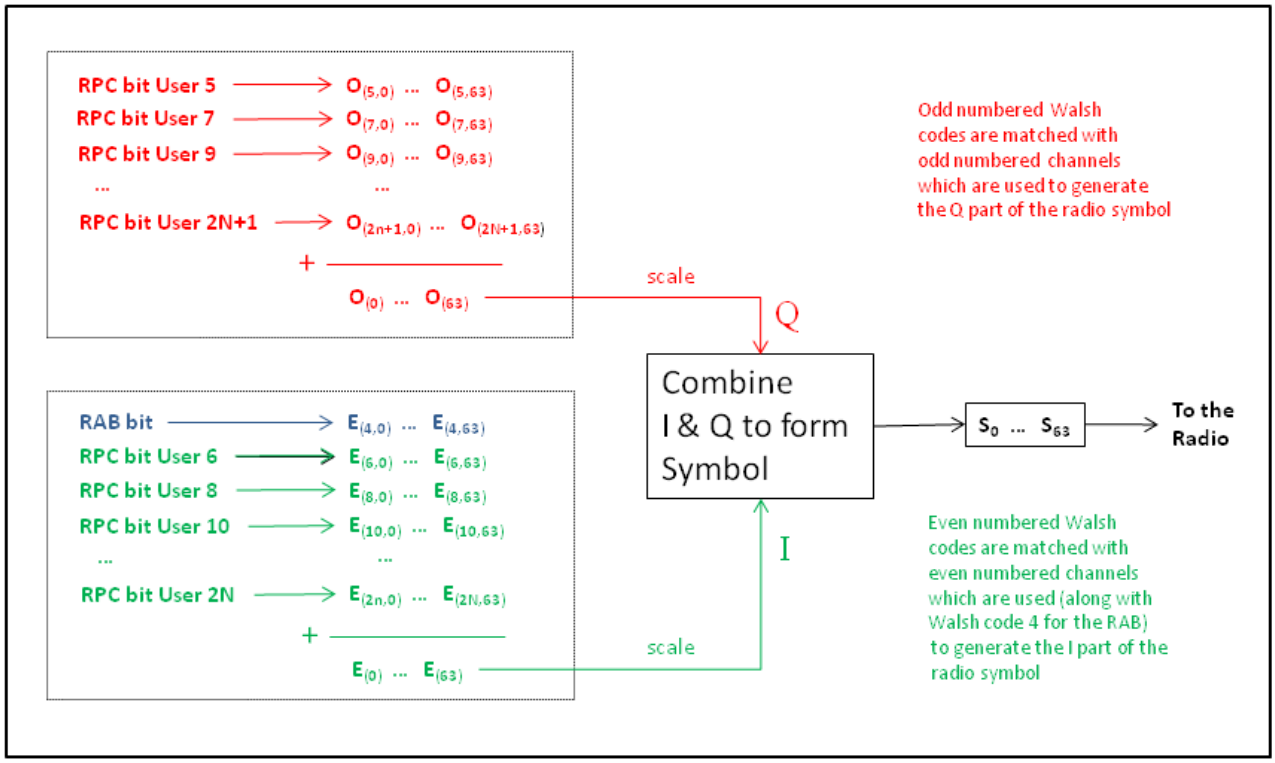

Figure 9 (940,400)

First, each bit of data is expanded into 64 chips using a $Walsh_{64}$ code. Walsh codes 0 to 3 are reserved. Walsh code 4 is used for the RAB bit. Walsh codes 5-63 are available for active connections.

Second, all the chips from even numbered Walsh codes are summed together and scaled. Then the same thing is done for the chips from the odd numbered Walsh codes.

Finally, the even and odd summed & scaled values are paired together to create radio symbols. So those N+1 bits become 64 radio symbols.

The 64 radio symbols will be repeated 4 times in each slot (IE a total of 256 symbols in each sot are dedicated to MAC data). The repetition increases the chances that the target will be able to successfully extract the desired bits.

The process for the DRCLock data is exactly the same, except that the source bits are the DRCLock data rather than the RPC data.

Pilot

We've discussed how control/user data gets converted to radio symbols. We've also discussed how MAC data gets converted to radio symbols. Now let's do the easiest case of all – we need to convert the pilot "data" to radio symbols.

The pilot consists of nothing but 0s. We take each 0, and expand it to 64 chips using Walsh_{64} code 0. We take each of those chips as the I part of a radio symbol. For the Q part we use the value 0. Thus we convert a single bit with value 0 into 64 radio symbols. We do that 3 times for each slot (because 192 radio symbols per slot are reserved for the pilot: 3 * 64 = 192).

Details of the Slot and Sub-Slot

So we now know how to convert any sort of data (IE bits) into radio symbols (IE (I,Q) pairs). What do we do with those radio symbols? We stuff them into slots! Each slot consists of 2 identical sub-slots. This is shown in Figure 10.

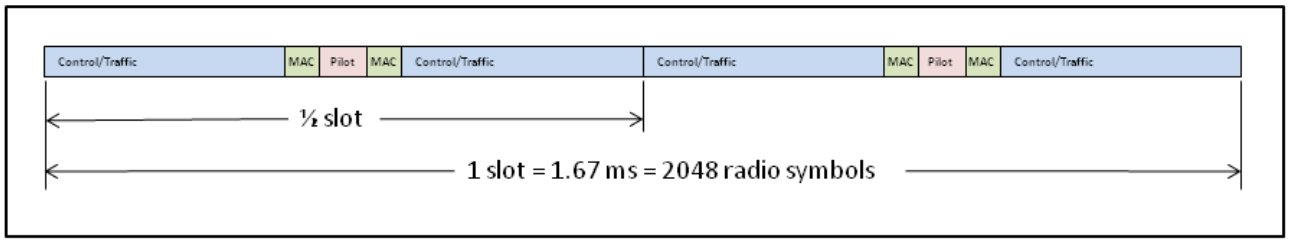

Figure 10 (920,220)

As shown in the diagram, each sub-slot consists of the following:

400 symbols of control/traffic data
64 symbols of MAC data
96 symbols of Pilot data
64 symbols of MAC data
400 symbols of control/traffic data

Doing the math, note that in each slot we have:

1600 control/traffic symbols (2 * 2 * 400)
256 symbols of MAC data (2 * 2 * 64)
192 symbols of pilot data (2 * 96)

Figure 10 shows what a slot looks like when there is user or control data to send. Figure 11 shows what a slot looks like when there isn't any such data to send:

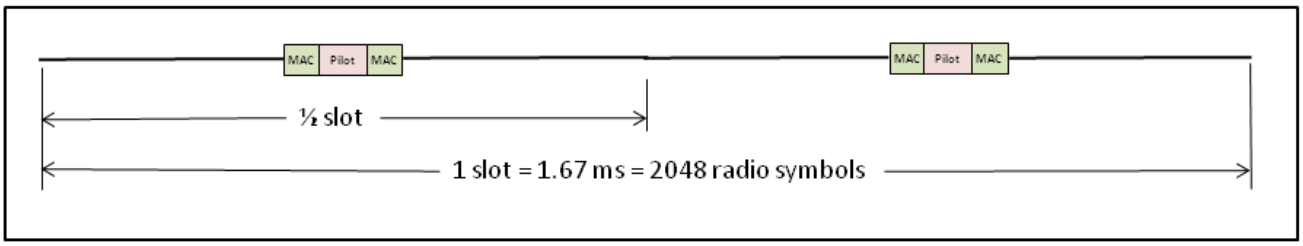

Figure 11 (910,210)

The above diagrams indicate that the Pilot and MAC data are always sent, in every slot. Control/User data is sent only when available.

Finally – the Big TDM Picture

We are now in a position to describe the TDM aspect of the EvDO Forward Link in all its glory. Look at Figure 12:

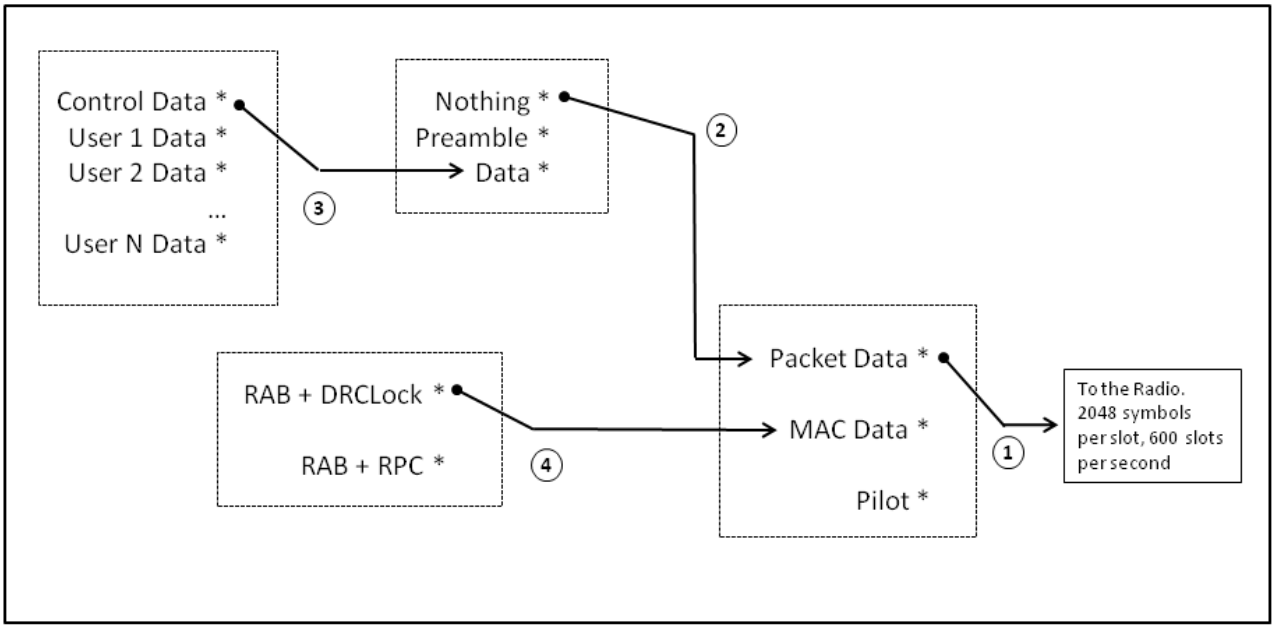

Figure 12 (950,500)

It is easiest to understand this figure from right to left – think of it as the radio "sucking" symbols from various sources. There are 4 switches:

Switch 1: This is a fixed, deterministic switch. First, grab 400 symbols of Packet Data, then 64 symbols of MAC data, then 96 symbols of Pilot, then 64 symbols of MAC data, then 400 symbols of Packet Data, REPEAT FOREVER.

Switch 2: For Packet Data we will have either real data to send, or we are just starting a packet and have to send prefix symbols, or … there might not be anything to send! In that case, we grab "nothing" symbols.

Switch 3: If switch 2 wants data, switch 3 decides whose data should be sent.

Switch 4: As noted earlier, for any given slot, we send either DRCLock data or RPC data. This switch decides which.

Note that switches 1 and 2 can change positions within a slot, while switches 3 and 4 only change on slot boundaries.

Recap

Let's walk through the whole thing again, starting with the radio interface.

The interface to the radio is actually a digital serial link, over which we send data at a rate of 14,745,600 bits per second. Those bits are grouped together in units of six, each group representing a signed integer – so we're sending 2,457,600 signed integers per second. Those integers are grouped together by pairs, each pair making up a "radio symbol", so we're sending 1,228,800 radio symbols per second, just as we described earlier.

Where do those radio symbols come from? They come from:

1) User Data - destined for actual people

2) Control Data – generated by other layers in the protocol stack

3) The "Packet Prefix" – one of 64 different, but pre-defined, bit patterns

4) The "Pilot Signal" – an unending supply of 0's

5) MAC data – "internal" data, dynamically generated, based on what's happening with the airwaves.

Each of the above data streams (of data bits) has a well defined (but different) process for getting itself turned into radio symbols.

Finally, there is a well defined algorithm for how the above data streams "take turns" getting sent to the radio.

The Control Channel Cycle

Up till now we've been primarily concerned with how things work at the slot layer and lower. Let's take a quick overview of the higher level structure.

16 frames constitute a "Control Channel Cycle", and this gives structure to how control information is sent out. A "Control Channel Capsule" consists of 1 or more control channel packets, and a control channel capsule will always be sent out at the start of a control channel cycle, and is called a "synchronous control capsule". Control channel capsules may also be sent out at other times, in which case they are called "asynchronous control capsules".

This is illustrated in Figure 13.

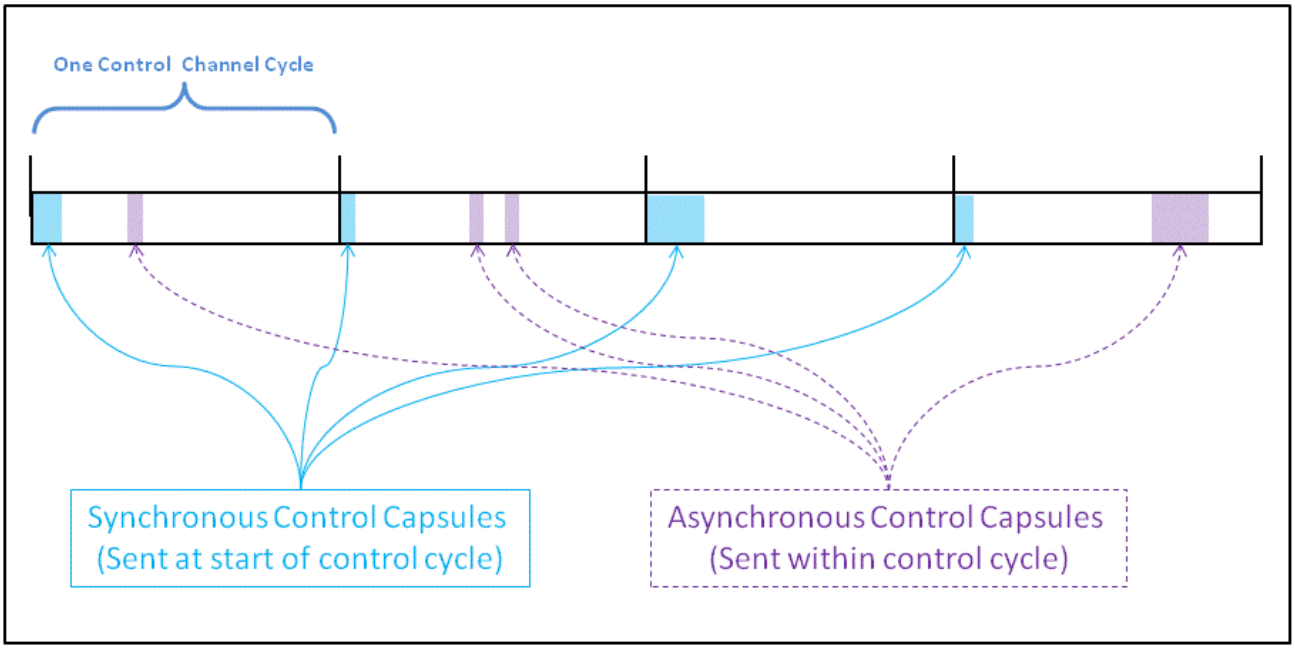

Figure 13 (930,500)

Since a Control Channel Cycle consists of 16 frames, that means it is made up of 256 slots, and that it's time duration is 426.67 milliseconds.

Note that the size of the Synchronous Control Capsule is variable – depending on how much data there is to send. Also, note that the Asynchronous Control Capsule is optional – if there is no data to be sent, then there is no Asynchronous Control Capsule.

The Sleep Cycle

12 Control Channel Cycles make up a Sleep Cycle (so each sleep cycle lasts 5.12 seconds). The cell tower always sends out pages at the beginning of a Control Cycle (IE in a Synchronous Control Capsule). Each AT will have a particular control cycle it will listen to within the sleep cycle. Thus an idle AT only has to wake up once every 5.12 seconds to see if anyone is trying to call it – thus increasing battery life.

This is illustrated in Figure 14, where AT 1 is listening for pages in the second control cycle, and AT 2 is listening for pages in the fifth control cycle.

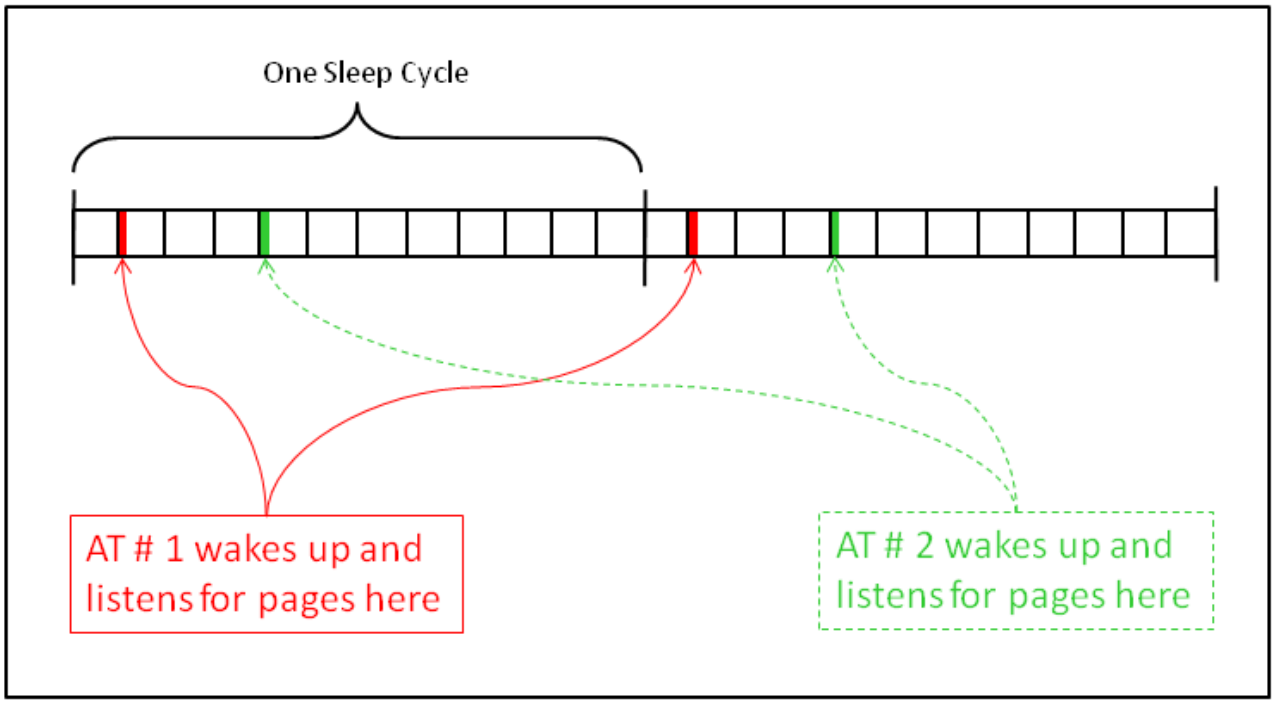

Figure 14

(770,460)