

Bell 202 Modem Object

©2009 Philip C. Pilgrim, Bueno Systems, Inc. (propeller@phipi.com)

This document and the software it describes are provided under terms of the MIT license, as follows:

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

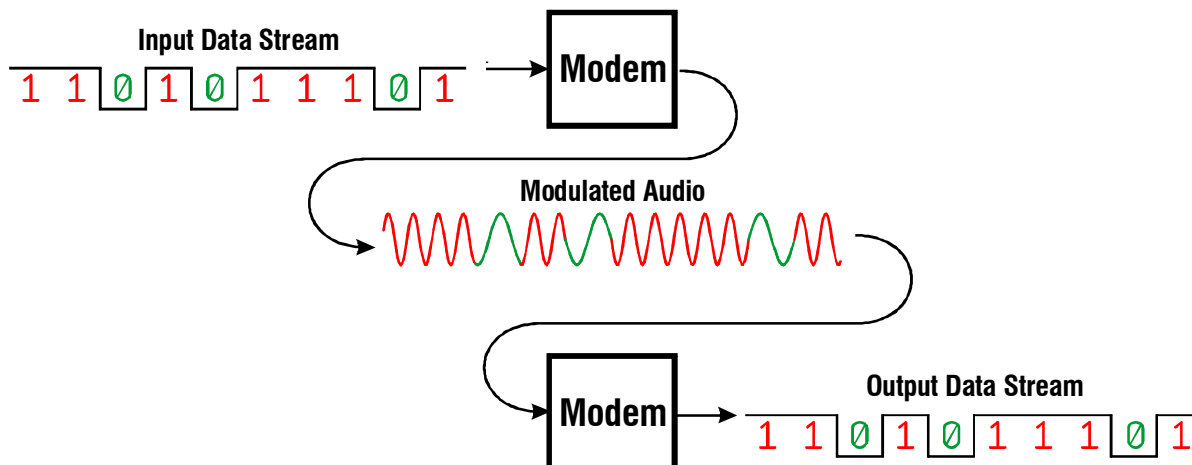
The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Introduction

The Parallax Propeller chip is capable of both generating and processing analog audio signals. This makes it particularly appropriate for transmitting and receiving data over audio channels. In such applications, the outgoing data is **modulated** at audio frequencies, and the incoming audio at the receiving end is **demodulated** in order to recover the original data. A program or device which performs these operations is thus called a **modem**.

When dial-up internet connections predominated, PC-hosted modems were quite common (and in some places still are). These are typically capable of transmission at 56K baud (bits per second) over regular phone lines. But many years prior to the development of these high-speed modems, slower speed devices using different standards were available. One such standard is known as "Bell 202". This is a half-duplex (transmissions occurring in one direction at a time) protocol operating at 1200 baud. Serial data consisting of ones and zeroes are coded as sine waves of 2200 Hz for "marking" (ones) and 1200 Hz for "spacing" (zeroes). This type of modulation is called "audio frequency shift keying" (AFSK). It is the modem's job to create the frequency-shifted sine waves at one end from the binary input data stream and to interpret the sine waves at the other end in order to reconstruct the same binary data stream at the output, hopefully without introducing errors. Here is what a typical data transmission might look like:

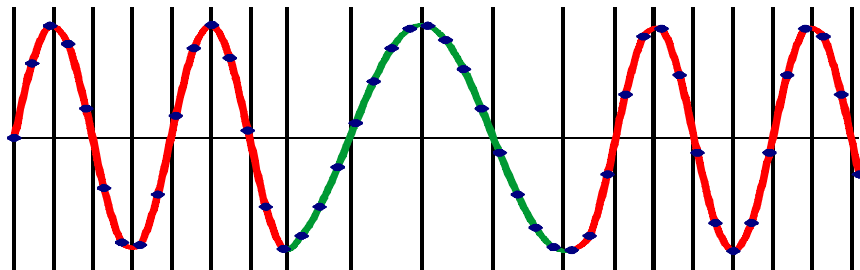


The medium through which the audio travels can be a wire pair, such as an audio cable, or the “ether”, as a radio transmission. The Bell 202 standard is frequently used among amateur radio operators to send data on VHF (very high frequency) bands using FM (frequency modulation). In fact the impetus for this project came from Ken Gracey at Parallax, who was using a pair of two-meter ham transceivers to exchange data with a mobile robot. As a result, much of the emphasis in this paper will center on the software’s use with amateur two-way radios.

Modulation

Modulation involves reading an incoming bitstream and producing a segment of 2200 Hz sine waves for each “one” bit, and a segment of 1200 Hz sinewaves for each “zero” bit. Since the output needs to be 1200 baud, each bit segment will be 1/1200th second long. This means that each “zero” segment will contain one full cycle at 1200 Hz, and each “one” segment will contain $2200 / 1200 = 1.833$ cycles at 2200 Hz. In the **bell202_modem.spin** object written for the Propeller, each bit segment is further divided into 16 sub-segments, for each of which an instantaneous output value is computed.

One further requirement is that the output waveform has to be continuous. This means that when the frequency changes, say, from 2200 Hz to 1200 Hz that the 1200 Hz waveform has to pick up at the same relative phase point where the 2200 Hz waveform left off. This is done to ensure that there are no high-frequency glitches in the output, as there would be from sudden discontinuities. The following diagram illustrates this requirement:



The vertical lines are at 90° phase intervals, *relative to the frequency of the waveform at that point*. Notice that, even though each waveform segment consists of 16 equally spaced points, the phase angle changes more rapidly in the 2200 Hz sections than in the 1200 Hz section. When we generate such a waveform, we will produce 16 points, equally-spaced in time; but the phase angle that we use to compute the sine at each point will advance at different rates for the high- and low-frequency segments. Moreover, when the instantaneous frequency changes, we will change the phase increment but continue from the actual phase at which the previous segment ended.

The Propeller includes a sine table in ROM, beginning at \$E000, that has 2049 entries covering a 0° to 90° interval. For the remaining 270°, the following trigonometric identities can be used to compute $\sin(x)$ from the table lookup function $\text{Sin}()$.

$\sin(x) = \text{Sin}(x)$	$0^\circ \leq x \leq 90^\circ$	(Quadrant I)
$\sin(x) = \text{Sin}(180^\circ - x),$	$90^\circ \leq x \leq 180^\circ$	(Quadrant II)
$\sin(x) = -\text{Sin}(x - 180^\circ),$	$180^\circ \leq x \leq 270^\circ$	(Quadrant III)
$\sin(x) = -\text{Sin}(360^\circ - x),$	$270^\circ \leq x \leq 360^\circ$	(Quadrant IV)

These identities are encapsulated in the following Propeller assembly code, which computes both sines and cosines over the entire 0° (\$0000_0000 internally) to nearly 360° (\$FFFF_FFFF internally) interval:

```

'-----[Cosine and Sine lookup routines]-----
' On entry, acc contains the angle: 0 to $ffff_ffff (unsigned).
' On exit, acc contains the sine or cosine: -$ffff to $ffff.

cosine      add      acc,_0x4000_0000      'Add 90 degrees for cosine.

sine        test     acc,_0x4000_0000 wz    'nz = quadrants II or IV.
            test     acc,_0x8000_0000 wc    'c = quadrants III or IV.
            shr      acc,#18               'Setup to index into ROM sine table.
            negnz     acc,acc               'Negate offset if quadrants II or IV.
            or        acc,sine_table        'OR offset to index.
            rdword    acc,acc               'Read the sine value into acc.
            negc      acc,acc               'Negate if quadrants III or IV.

sine_ret
cosine_ret  ret                            'Return value (-65535 - 65535) in acc.

```

Here is a Spin-like pseudo-code encapsulation of the modulation function used to produce one bit:

```

Case bit

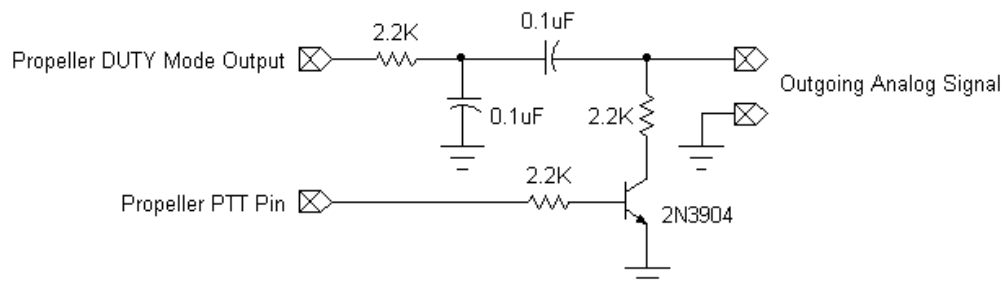
    0: PhaseInc := Inc1200
    1: PhaseInc := Inc2200

Repeat i from 0 to 15

    Level := sine(Phase)
    Phase += PhaseInc
    Wait 1/(1200 * 16) seconds
    Output(Level)

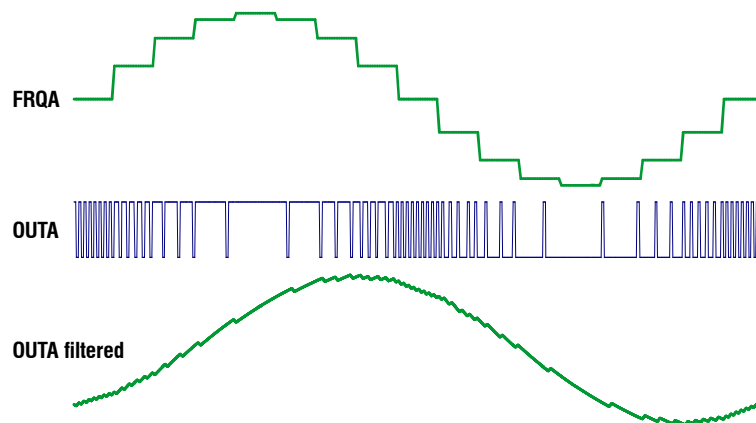
```

To produce an actual analog level, as implied by the above **Output** function, we need to employ one of the Propeller's counters in its DUTY mode. In this mode the counter, say **ctra**, adds the value in **frqa** to its **phsa** register on every clock cycle, producing a logic "high" on its output pin whenever a carry out of **phsa**'s most-significant bit occurs. This behavior has the following property: if **frqa** is low, carries will happen infrequently, so the relative number of highs versus lows on the pin will be low; if **frqa** is high, carries will happen frequently, producing more highs versus lows on the pin. By applying an RC low-pass filter to the output, a voltage proportional to the high-versus-low outputs and, hence, to the value of **frqa** can be obtained. It's also a good idea to add a series capacitor to AC couple the output to the audio channel. Here's a typical low pass filter schematic used with a two-way radio:



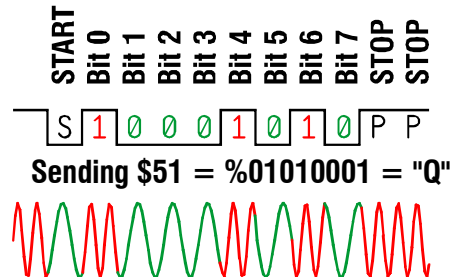
The schematic includes an extra transistor and two resistors, which combine to form a push-to-talk (PTT) trigger that will work with many two-way radios that lack a separate PTT input. This part of the circuit may be ignored for the purpose of discussing modulation.

Here is a visual representation of the effect produced by the DUTY mode output and low pass filtering:



Notice that the low-pass filtering produces a slight phase lag in the output. This is a normal side effect of a simple RC filter and won't cause us any problems. (Actually, since OUTA's duty cycles have a *much* higher frequency than what is illustrated here – by a factor of more than 1000 – the filtering and consequent phase shift are less severe than what was required to produce the illustration.)

So far, we've seen how to produce one bit's worth of modulated output. The next job is to string multiple bits into bytes so we can send some real data. To do this we will use the same asynchronous protocol used in common RS232 and other serial modes of communication. In this protocol, each byte of data is sent with a "0" bit preamble (the "start" bit), followed by eight data bits, least-significant bit first, and followed by one or more "1"s (the "stop" bits). A typical byte will look like this:

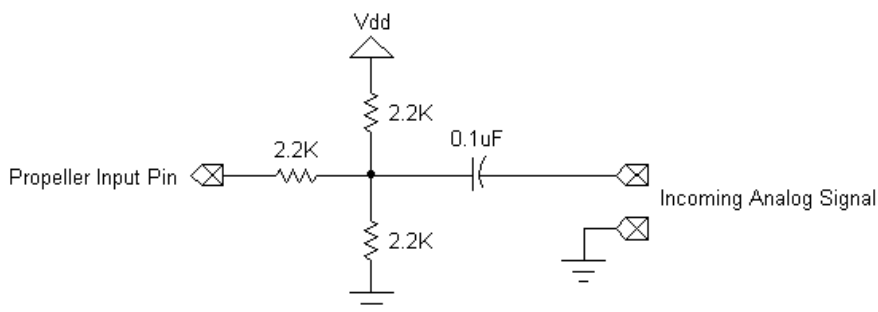


The minimum number of stop bits to use depends somewhat on the reliability of the data channel. Since the modem can be used for radio transmission, which is subject to noise and other kinds of interference, it's sometimes possible for the receiver to get confused if a noise blast should obliterate one or more bits. Sometimes, after a noise burst, the receiver will misinterpret a "0" data bit for a start bit and begin decoding a byte in the wrong place. Depending on the data, it could stay out of sync for several bytes before encountering a "0" stop bit, which is an error condition, forcing resynchronization. By adding extra stop bits, an out-of-sync receiver can come back into sync more quickly, since the bytes become slightly more separated from each other. We will use two stop bits, for a total of eleven bits per transmitted byte. Tests have shown that this increases reliability significantly over using a single stop bit.

Finally, since we're dealing with half-duplex transmission that requires a "turnaround" between transmitting and receiving, we need to inject a time delay at the beginning of each new transmission. We will do this by sending a continuous 2200Hz tone for a half second, which is the default "marking" or "1" state. This not only gives the other modem time to switch to "receive", but allows it to synchronize to the marking state.

Demodulation

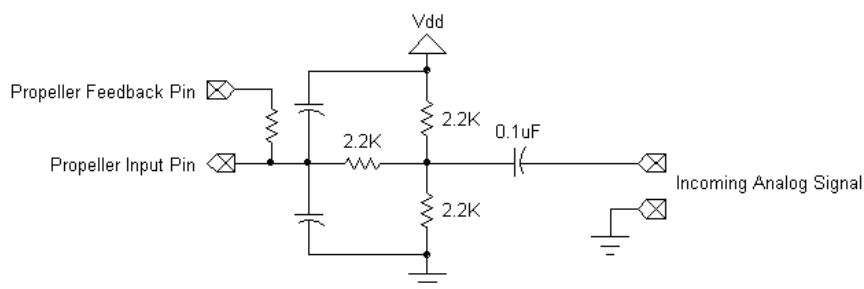
Demodulation is the process of taking a modulated signal like that produced above and converting it back to discrete bits. But first we have to convert the modulated signal, which is analog, to a digital representation the Propeller can handle. There are two ways to do this: 1) thresholding, and 2) analog-to-digital conversion. Thresholding is the simplest and can be accomplished with a minimum of external hardware. Three resistors and a cap are the only components required:



The cap is there to AC-couple the incoming signal to the Propeller. The voltage divider ensures that the AC-coupled signal straddles the Propeller's logic threshold equally, thus converting it to square waves. The additional series resistor is there to protect the Propeller from signal levels that exceed 3.3V peak-to-peak.

In operation, the input pin is not sampled directly by the demodulation code; rather, it is input to one of the Propeller's counters, which counts up by one at an 80MHz rate every time it sees a "high" on this input. At each sampling interval, the program reads the count value from the counter's **phsx** register and subtracts the previous value to get the "width" of the "high" during the current sampling interval. That way, there is less chance that a single errant reading will exert undue influence on the sample.

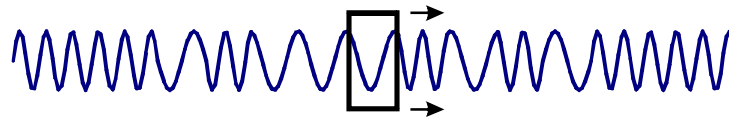
The Propeller is also capable of digital-to-analog conversion, using a "delta-sigma" technique. This requires an additional resistor and, ideally, a couple more caps (0.001uF or so):



This technique also requires a counter that counts up every time it sees a "high" on its input pin. But this time the counter fights back by returning the *opposite* signal on a feedback pin. The net effect when the feedback is coupled to the input pin is to keep the input pin biased at the Propeller's logic threshold of around $V_{dd}/2$. But here's the interesting part: the higher the incoming voltage is, the more times during the sample interval the input pin will be above threshold and have to be corrected. What this means is that the number of new **phsx** counts during the sampling interval will be proportional to the average input voltage during that interval. The constant of proportionality will depend on the value of the feedback resistor, relative to the (nominal) 2.2K input resistor, which sets the A/D converter's "gain".

So now we have a way to read an input signal, be it a simple sample count or an average voltage. Both methods involve reading the value from a counter's **phsx** register, which makes the demodulator code universal in the sense that, once the counter is configured, it needn't know how the data are being obtained.

To perform the actual demodulation, we not only have to recognize the difference between the 1200 Hz and 2200 Hz segments (slicing), but we also have to locate the boundaries between them accurately (dicing). So, to get on with all this slicing and dicing, we might be tempted to ask, "What is the frequency at each point of the incoming waveform?" After all, if we knew that, we could assign each point a **0** or a **1**, based on whether the frequency was 1200 Hz or 2200 Hz. Unfortunately, frequency is not something that can be defined at individual points but, rather, over an interval of points. So what we're going to do instead is take a window, one bit-time ($1/1200^{\text{th}}$ second) wide and slide it across the incoming waveform, incrementing $1/16^{\text{th}}$ of one bit time at each step:

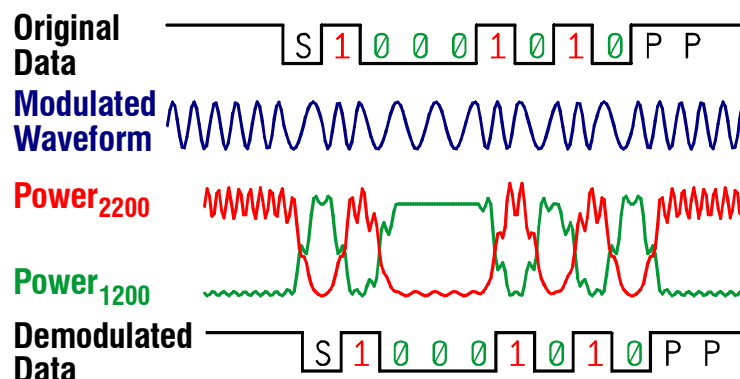


At each position, we will analyze the waveform inside the window for its 1200 Hz and 2200 Hz frequency components and gauge their relative amplitudes. Windows in which 1200 Hz dominates will be deemed "more zero than one", and *vice versa* for windows in which 2200 Hz dominates.

To get these relative amplitudes at each window position, given the incoming modulated signal samples **Sig_i**, we will compute the *Fourier power coefficient* for each frequency: 1200 Hz and 2200 Hz. This is given by:

$$\text{Power}_f(t) = \left[\sum_{i=t-15..t} \text{Sig}_i \cdot \sin(2\pi \cdot i/16 \cdot f/1200) \right]^2 + \left[\sum_{i=t-15..t} \text{Sig}_i \cdot \cos(2\pi \cdot i/16 \cdot f/1200) \right]^2$$

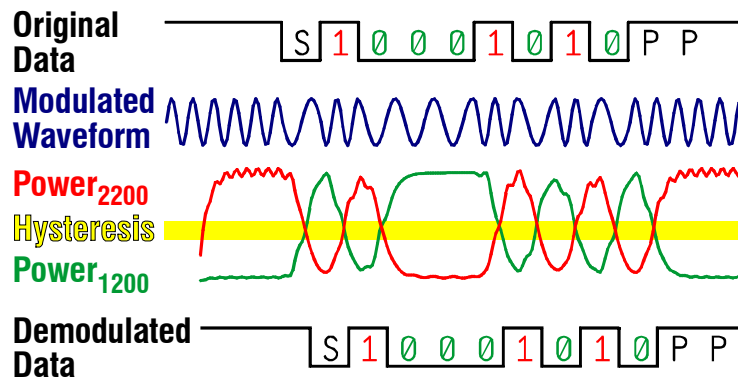
By comparing **Power₁₂₀₀(t)** with **Power₂₂₀₀(t)** at each incremented window position **t**, we will be able to tell whether the waveform fragment inside the window is "more one than zero" or "more zero than one", and we can assign a corresponding binary value to that window position. Here's are graphs of Power₁₂₀₀ and Power₂₂₀₀ for the waveform shown in the previous illustration:



Note that this demodulation introduces a half bit's worth of delay between the original binary data and the demodulated binary data.

In the Bell 202 object, the power functions go through an additional level of smoothing to filter out any high-frequency glitches. After that, the comparisons between Power₁₂₀₀ and Power₂₂₀₀ are performed with a programmable amount of hysteresis. "Hysteresis", in this case, means that a $0 \rightarrow 1$ or $1 \rightarrow 0$ transition cannot take effect until the absolute difference between Power₁₂₀₀ and Power₂₂₀₀ is greater than a certain threshold. This creates a "hysteresis band", inside of which no transitions can take place. This helps to ensure that the transitions between "0" and "1" bits are clean, without bouncing rapidly back and forth

near the threshold in the case of noisy data. The following illustrates the effects of these additional measures. The yellow region is the hysteresis band:



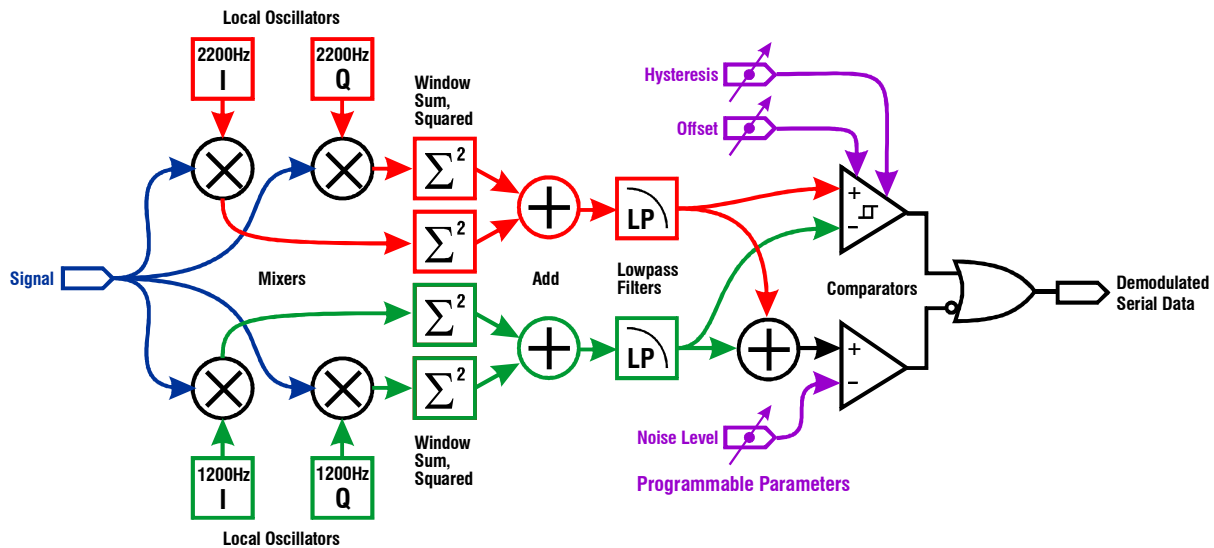
One final adjustment needs to be made before comparing Power_{1200} and Power_{2200} : the application of a “slicing” or “offset” level. This is a value that’s added to Power_{2200} before Power_{1200} is compared with it. It’s necessary because the modulator’s RC filter – not to mention the audio data channel – will attenuate the 2200 Hz frequency more than the 1200 Hz frequency. This will result in a narrowing of the “1” bits relative to the “0” bits, which could lead to received data errors. By engaging a programmable slicing level, this disparity can be corrected.

Another problem that can arise on a radio channel is what to do when there’s no signal present. In such a case, comparing anything computed for Power_{1200} and Power_{2200} makes no sense and could lead to garbage data. The way around this is to *add* these two components and to see whether, combined, they are higher than a programmable noise threshold. If not, any data derived from them will be discarded. This is similar to the way the squelch controls on many radios work.

As a final guard against garbage data, we keep track of how many “0” and “1” samples are detected in each 16-sample bit window. For example, if the ratio were 9:7, that’s not convincing enough to assign a bit value one way or the other. In the Bell 202 object, the ratio between the “winner” and the “loser” has to be at least 11:5 to count as a legitimate bit.

To summarize, we’ve introduced a couple strategies for detecting errors: 1) Comparison of the signal to a noise threshold to make sure it’s higher, and 2) requiring a supermajority of at least 11:5 for a bit value to be assigned, given a 16-count sample. If either condition is not met during demodulation, the bit decoding process is reset, and the receiver is forced to wait for the next start bit.

What we have so far can be described schematically, as if it were a hard-wired circuit. The multiplication of the incoming signal by sine (called “I” or “in-phase”) and cosine (called “Q” or “quadrature-phase”) oscillations is often performed in hardware by analog multipliers, or “mixers”. The comparison operation could be performed in hardware by a comparator with adjustable hysteresis and offset levels. Here is a block diagram that summarizes (ignoring the error reset) what the Propeller does in software:



Using the Modem Object

Like most Propeller objects, the modem object requires calling a “start” method to initialize everything and get it going. Unlike most other objects, this object has three start methods. Which one you use depends on your particular hardware setup. The following sections describe each one.

Also, this object requires another object, **umath.spin**, which is included in this bundle and which is also available separately from the Propeller Object Exchange.

Note: In all the following example code fragments, it’s assumed that the modem object has been given the name **mdm** in a prior **OBJ** declaration:

```
OBJ
mdm : "bell202_modem"
```

start_simple

Usage: **start_simple(rcvpin, xmtpin, pttpin, mode)**, returning new cogid + 1 on success; 0 on failure.

This is the start routine to call if your transmit circuit (non-feedback version) looks like those in the Modulation and Demodulation sections of this document. All you have to specify are the receive, transmit, and PTT pins. Any of these could be set to **NONE** if you’re only transmitting, only receiving, or not using PTT. Finally, the mode parameter can be used to set the transmit audio level (**LVL0** through **LVL8**) and can be bit-wise ORed (or not) with **AUTOXR**, which will automatically switch channel directions when you begin transmitting or receiving. If **mode** is zero, it will default to **LVL6 | AUTOXR**. When **start_simple** returns, the modem will be in **standby** mode. Here’s an example that sets up the modem object to receive on pin A5, transmit on pin A6, with no push-to-talk, and a transmit level of 6. Then **receive** is called to begin receiving right away.

```
mdm.start_simple(5, 6, mdm#NONE, mdm#LVL6)
receive
```

Note: You should experiment to get the optimum transmit level. If you’re using a PTT circuit multiplexed to the microphone input (like the one shown in this document) and the volume is set too high, it can interfere with the PTT, causing intermittent transmission.

start_bp

Usage: **start_bp(mode)**, returning new cogid + 1 on success; 0 on failure.

This start routine is used with Parallax's "Propeller Backpack" board (#28327), which is preconfigured with all the passive components required by the modem. The only option is the **mode** parameter, which is the same as for **start_simple**. When **start_bp** returns, the modem will be in **standby**. Here's some sample code that sets the output to level 5 and enables automatic transmit/receive. Receiving would then begin as soon as **receive** or one of the input methods is called. Likewise, transmitting would begin as soon as **transmit** or one of the output methods is called.

```
mdm.start_bp(mdm#LVL5 | mdm#AUTOXR)
```

start_explicit

Usage: **start_explicit(rcvm, rfbm, rhim, rlom, xmtm, xhim, xlom, shim, slom, mode)**, returning new cogid + 1 on success; 0 on failure.

This is the mother of all start methods and the one the other two invoke to get their work done. It should work with virtually any hardware configuration. Excepting the **mode** parameter (explained above), all the other parameters are bit masks that are used to select pins on port A. A "1" in any bit position selects the corresponding pin as having a role for the particular parameter. The parameters are as follows:

rcvm: Receive pin mask. Contains a **1** at the bit position corresponding to the pin that receives the incoming audio. May be set to zero if modem is transmit-only.

rfbm: Receive feedback mask. Contains a **1** at the bit position corresponding to the pin that provides counter feedback to **rcvm** for delta-sigma A/D conversion. May be set to zero for non-analog input schemes.

rhim: Receive high mask. Contains **1s** at the bit positions for every pin that must be set to output a logic "high" during receive. For example, a pin-controlled voltage divider leg can be left floating during idle or transmit periods to reduce overall current consumption.

rlom: Receive low mask. Contains **1s** at the bit positions for every pin that must be set to output a logic "low" during receive.

xmtm: Contains a **1** at the bit position corresponding to the pin that is used to provide the DUTY mode audio output. May be set to zero if modem is receive-only.

xhim: Transmit high mask. Contains **1s** at the bit positions for every pin that must be set to output a logic "high" during transmit.

xlom: Transmit low mask. Contains **1s** at the bit positions for every pin that must be set to output a logic "low" during transmit.

shim: Standby high mask. Contains **1s** at the bit positions for every pin that must be set to output a logic "high" during standby.

slom: Standby low mask. Contains **1s** at the bit positions for every pin that must be set to output a logic "low" during standby.

mode: Sets the transmit audio level and auto transmit/receive flag, as defined above. This parameter is required. There is no default provided if it's set to zero.

Notice that there is no special mask for PTT. This can be established using **rlom**, **slo**m, and **xhim** for an active high enable or **rhim**, **shim**, and **xlom** for an active low enable.

Here's an example invocation in which pin A5 is used for input, A6 for feedback, A7 for transmit, and A8 for a high-enable PTT. The audio output level is 5, and there is no automatic transmit/receive switching.

```
mdm.start_explicit(|<5, |<6, 0, |<8, |<7, |<8, 0, 0, |<8, mdm#LVL5)
```

stop

Usage: **stop**

Calling **stop** halts the modem software and frees a cog.

receive

Usage: **receive**

This method waits for any data already buffered for output to be sent, if the modem is transmitting. Then it configures the selected port A pins and modem for receiving data. Received data will be buffered in the background for readout by the input routines.

transmit

Usage: **transmit**

If the modem is not currently in transmit mode, this method configures the selected port A pins, including any PTT pin, and modem for sending data. Then the modem waits for 1/2 second while the transmitter is sending "mark" 2200 Hz. This gives the receiver at the other end time to break out of squelch and adjust to the incoming audio.

standby

Usage: **standby**

If the modem is currently in transmit mode, this method waits for the transmit buffer to empty, then switches the modem into a neutral mode whereby it's neither transmitting nor receiving. With certain pin configurations, this mode can be used to save current consumption.

Note: None of the methods **receive**, **transmit**, or **standby** needs to be called if automatic transmit/receive is in effect, although they can be called if you want.

out

Usage: **out(char)**, returning **true** on success, **false** on failure.

Send the byte value **char**. If the modem is not in transmit mode and if automatic transmit/receive is in effect, it will enter transmit mode first. Once in transmit mode, this routine will block, waiting for available buffer space. Then, **char** is buffered for transmission. If, however, the buffer is already full and the modem is not transmitting, **out** will return **false**, indicating failure.

The following example sends an ASCII 13 (\$0D), a carriage return:

```
mdm.out(13)
```

outstr

Usage: **outstr(straddr)**, returning **true** on success, **false** on failure.

This method sends an entire zero-terminated string, whose *address* is given by **straddr**. It calls **out**, and switches modes (or not) accordingly. If any of its calls to **out** results in failure, **outstr** will abort and return **false**. Here's an example that sends the string "Testing 123":

```
mdm.outstr(string("Testing 123"))
```

inp

Usage: **inp**, returning a value.

This method will switch to receive if automatic transmit/receive switching is in effect. Then it will return the next character from the input buffer. If the receive buffer is empty but the modem is configured to receive data, **inp** will block until a character appears. Otherwise, it will return the constant **NONE** (\$8000_0000).

In this example, the program loops until it receives a question mark:

```
repeat until mdm.inp == "?"
```

inp0

Usage: **inp0**, returning a value.

This non-blocking input method will switch to receive if automatic transmit/receive switching is in effect. Then it will return the next character from the input buffer. If the receive buffer is empty, it will return the constant **NONE** (\$8000_0000).

In this example, **inp0** is called, and if a *bona fide* character was received, it's displayed on a TV monitor. Note that "<> **mdm#NONE**" can be replaced with "**=> 0**", since **NONE** is negative.

```
if ((char := mdm.inp0) <> mdm#NONE)
    tv.out(char)
```

inpstr

Usage: **inpstr(straddr, terminator, maxlen, maxtime)**, returning a value.

This method will switch to receive if automatic transmit/receive switching is in effect. Then it reads characters into the string (byte array) whose address is given by **straddr**. Reading continues until a character given by **terminator** is read or **maxlen** characters have been read, whichever comes first. In either case the string is terminated with a zero character. Therefore, the array at **straddr** must have a size of at least **maxlen** + 1. If you don't want a terminal character to be recognized, set **terminator** to **NONE**.

Optionally, if the **maxtime** argument is greater than zero and less than $2^{32} \cdot 1000 / \text{clkfreq}$ (53687 for an 80 MHz clock), the method will time out after **maxtime** milliseconds, returning the characters received up to the timeout.

The value returned by this method is the number of characters received, excluding the zero terminator (i.e. the resultant string length).

In the following example, the program waits for a command indicator ("!"), then reads a three-character command into the byte array **cmd**, waiting indefinitely, if necessary, since **maxtime** is zero:

```
repeat until mdm.inp == "!"
mdm.inpstr(@cmd, mdm#NONE, 3, 0)
```

waitstr

Usage: **waitstr(straddr, maxtime)**, returning **true** on success, **false** on timeout.

This method will switch to receive if automatic transmit/receive switching is in effect. Then it waits for a string of consecutive input characters matching the string at **straddr**. If **maxtime** is greater than zero and less than $2^{32} \cdot 1000 / \text{clkfreq}$ (53687 for an 80 MHz clock), the method will time out after **maxtime** milliseconds, returning **false**.

For best results, the string to be matched should not have any duplicates of its first character. The reason is that upon a mismatch, the match index is reset to zero. So, for example, if the string to be matched was "MAMA MIA" and the incoming string was "MAMAMA MIA", it would not match because, once **waitstr** saw the incoming "MAMAM", it would reset the pointer to the beginning of "MAMA MIA", which the remaining "AMA MIA" in the input stream would fail to match.

In this example, we are looking for the string "\$GPGGA", timing out after a two-second wait. If the "\$GPGGA" is received in time, the rest of the string, up to and including a carriage return (13), is buffered in **bytearray**.

```
if (mdm.waitstr(string("$GPGGA"), 2000))
    mdm.inpstr(@bytearray, 13, 80, 0)
else
    tv.outstr(string("No signal from GPS.", 13))
```

outchars

Usage: **outchars**, returning a value.

Retrieve the number of characters in the output buffer awaiting transmission. The following example uses **outchars** to ensure that a one-second delay is inserted between message sections:

```
mdm.outstr(string("This is a very long message to fill the buffer. Now wait a second..."))
repeat while mdm.outchars
waitcnt(cnt + clkfreq)
mdm.outstr(string("Okay, that was one second."))
```

inpchars

Usage: **inpchars**, returning a value.

Retrieve the number of characters in the input buffer. **Note:** Calling **inpchars** does *not* start **receive** automatically when automatic transmit/receive is in effect. Therefore, any **repeat** loop begun with the modem in **transmit** or **standby** that waits for **inpchars** to be non-zero will never terminate. To switch automatically to receive under **AUTOXR**, you must call **inp**, **inp0**, **waitstr**, or **inpstr**. Of course, you can always just call **receive** first, as well.

In the following example, the program will not input the next four characters unless there are at least that many in the buffer available to be read:

```
if (mdm.inpchars => 4)
    inpstr(@strarray, 4)
```

set

Usage: **set(param, value)**

This method sets any of three demodulator variables after the modem has been started. The **param** argument specifies which variable to set and can be one of **HYST** or "**H**" (hysteresis), **SLICE** or "**S**" (0/1 slicing/offset level), and **NOISE** or "**N**" (noise/ squelch level). The **value** parameter is the new value for the selected variable. All three values default upon start to usable levels, but it may be necessary to tweak them for optimum performance. The provided programs, **modem_monitor.exe** and **modem_monitor.spin**, can be used to do this accurately and efficiently. See the **Monitor Program** section for details.

In this example, the hysteresis is set to zero:

```
mdm.set(mdm#HYST, 0)
```

signal

Usage: **signal**, returning a value.

Return latest instantaneous signal stats from the demodulator in the form:

%zLLLLLLLLLLLLL_oHHHHHHHHHHH_nnnnnnnn, where

z and **o** (bits 31 and 19) are coded as follows:

- 0 0** : Signal is below noise threshold.
- 0 1** : Signal is marking (1).
- 1 0** : Signal is spacing (0).
- 1 1** : Signal is in hysteresis band between mark and space.

L...L	(bits 30 .. 20)	sum-of-squares of 1200 Hz demodulator outputs.
H...H	(bits 18 .. 8)	sum-of-squares of 2200 Hz demodulator outputs.
n...n	(bits 7 .. 0)	-16 (pure 0) to 16 (pure 1) level count for latest bit (2's complement).

This method is used extensively in **modem_monitor.spin** to return signal quality information to the PC host program. It is probably not of much use beyond that, except possibly for auto-tuning (left as an exercise for the reader). **Note:** This method always returns the last values obtained during reception *and does not reset them*. For this reason, if the reception of active data ceases, it will continue to return the same old values.

Monitor Program

The Windows program **modem_monitor.exe** comes bundled with this object. It is used to tune the modem parameters for optimum data reception. Also included in this bundle is the Propeller-resident monitor program, **modem_monitor.spin**, listed here:

```
CON

_clkmode      = xtall + pll8x      '<-----Change this line, as appropriate, for your
Propeller setup.
_xinfreq      = 10_000_000        '<-----Change this line, as appropriate, for your
Propeller setup.

VAR

    byte  inpstr[5]

OBJ

    mdm   : "Bell202_modem"
    dbg   : "FullduplexSerial"

PUB  Start | nstr, ch, param

    dbg.start(31, 30, 0, 38400)
    mdm.start_bp(0)                '<-----Change this line, as appropriate, for your
modem setup.
    mdm.receive
    nstr~
    waitcnt(cnt + clkfreq / 2)
    repeat
        if (mdm.inpchars)
            dbg.tx("?")
            dbg.tx(mdm.inp)
            dbg.tx("!")
            dbg.hex(mdm.signal, 8)
            dbg.tx(10)
            if ((ch := dbg.rxcheck) == 0)
                if (nstr and nstr < 5)
                    inpstr[nstr++] := ch
                    if (nstr == 5)
                        mdm.set(inpstr[1], hex(inpstr[2])<<28 | hex(inpstr[3])<<24 | hex(inpstr[4])<<20)
                        nstr~
                    elseif (ch == "!")
                        nstr := 1

PRI  hex(char)

    if ((char -= "0") > 9)
        char -= 7
    return char
```

You will probably have to change some lines in this program (marked above) to get it to work with your particular setup. Then, you will need to connect the audio input to a radio or other audio source.

You will also need a transmitter running full bore sending data for the monitor program to pick up. Something like the following will work fine. Just be sure to send a variety of data, not just the same character repeated *ad infinitum*.

```

CON

_clkmode      = xtall + pll8x  '<-----Change this line, as appropriate, for your setup.
_xinfreq      = 10_000_000    '<-----Change this line, as appropriate, for your setup.

OBJ

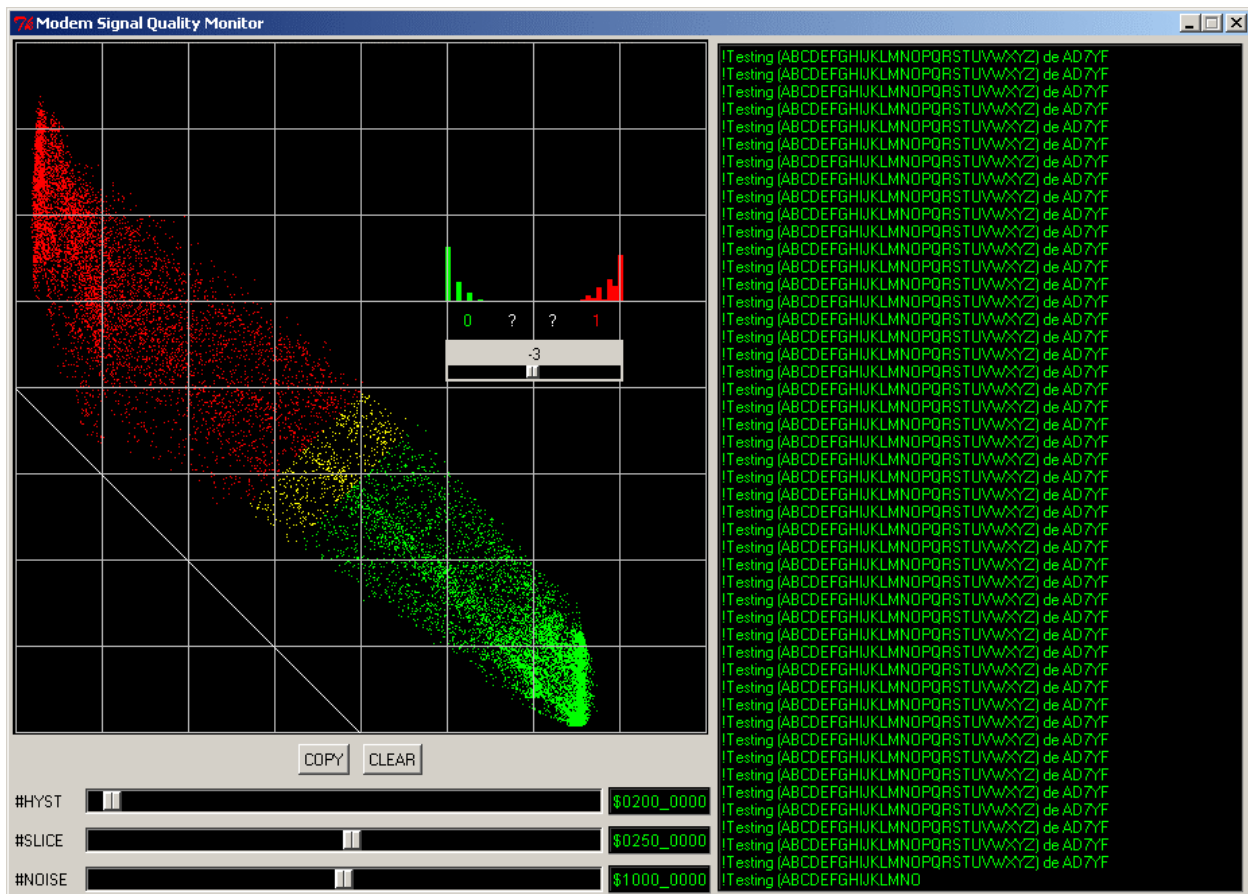
mdm : "bell202_modem"

PUB Start

mdm.start_bp(0)                '<-----Change this line, as appropriate, for your setup.
waitcnt(cnt + clkfreq / 2)
repeat
  mdm.outstr(string("!Testing "))
  mdm.outstr(string("ABCDEFGHIJKLMNOPQRSTUVWXYZ"))
  mdm.outstr(string(" de YOUR CALLSIGN", 13)) '<-----Add your callsign here for a radio.

```

Once you've got the transmit and receive ends set up and working with a clear data channel, make sure your receiver's program port is connected to your PC, and start **modem_monitor.exe**. It will search the available serial ports for output from the receiver. Assuming it finds the expected data patterns, it will display the following:



The right-hand panel shows how the modem is currently interpreting the incoming signal. You can compare this for accuracy with what you've programmed the transmitter to send.

The scatter plot shows the low-pass-filtered 2200 Hz sum-of-squares response (y axis) graphed against the same for 1200 Hz (x axis). Red dots correspond to those samples deemed "1"s; green, for "0"s. The yellow dots are in the hysteresis band, so may have been interpreted either way. You can adjust the width of this band in real time by moving the **#HYST** slider. Ideally, all the red, yellow, and green dots would lie along a single, thin diagonal line. In the real world, there will be some spread, as the above screen capture illustrates.

The thin, white diagonal line represents the current noise threshold. Anything above the line is deemed "signal"; below it, "noise". By moving the **#NOISE** slider, you can adjust the position of the noise threshold. Don't move it too close to the signal dots, though, particularly if your transmitter and receiver are in close proximity. As your communications channel becomes less reliable with distance, these dots will naturally migrate lower, and you don't want them misinterpreted as noise if some sense can be wrested from them.

Each received bit is sampled sixteen times. Within each bit frame, the modem program counts how many times the sum-of-squares comparisons produced a "1" level and how many times a "0" level. Then it's just a matter of voting to get a bit value, but it takes a super majority to "win". The little bar graph embedded in the scatter plot keeps track of how many times a particular vote count occurred for each bit. Ideally, only the bars at the very ends would show any counts, signifying unanimity. But unanimity is hard to come by in the real world, as the plot above shows. What's important, then, is that the average vote count for "0" equals the average vote count for "1". The number in the little slider below the bar graph shows the difference between these. You should adjust the **#SLICE** slider to get this number as close to zero as possible. At this point the "1" versus "0" detection will be balanced.

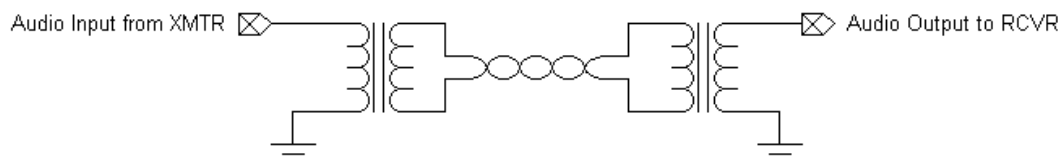
You can clear the display at any time by clicking the "CLEAR" button. Once you've made your adjustments to the demodulator settings, you can click the "COPY" button to save them to the clipboard. When you paste the settings into your programs that use the modem object, you will get something like this:

```
mdm.set(mdm#HYST, $0200_0000)
mdm.set(mdm#SLICE, $0250_0000)
mdm.set(mdm#NOISE, $1000_0000)
```

This code should be pasted right after your call to the modem's start routine, whichever one you use. (Make sure to indent it properly!)

Audio Channel Hardware

Although the thrust of this discussion has centered upon radio communication, the same techniques can be applied to hard-wired audio connections. For best results over long distances, isolation transformers (baluns), along with a twisted-pair cable should be used, as the following schematic illustrates:



The **bell202_modem** object is designed for half-duplex operation. By starting two separate instances of the object, however, full-duplex operation is possible, with each instance handling traffic in a single direction. Therefore, a hard-wired full duplex system would involve four transformers and two twisted pair cables. (There is a possibility that a two-transformer/single-twisted-pair arrangement could be used instead, along with a two-wire to four-wire "hybrid" circuit; but as of this writing, this has not been tried.)

Regulatory Issues

In the United States, communication via the airwaves and public wire networks is regulated by the Federal Communications Commission (FCC). Any connections made to the public telephone system, for example, must be done using an FCC-approved Data Access Arrangement (DAA). This is to protect the phone system from damage caused by poorly-designed or -implemented hardware and is governed by Part 68 of the FCC regulations. There may be other requirements as well, which are beyond the scope of this document to consider.

Communication via amateur radio is governed by Part 97 of the FCC regulations and requires a license. A Technician Class license is very easy to obtain and is a highly recommended route for the kind of experimentation this modem object enables.

Data transmission via unlicensed radio services (e.g. Family Radio Service) is either heavily restricted or prohibited altogether. Be sure to read the appropriate regulations carefully before even contemplating unlicensed operation.