QEX

$5

January/February 2016

www.arrl.org

A Forum for Communications Experimenters

Issue No. 294

PY2EAJ shows us how to use digitally tunable capacitors to build a band-pass filter that employs a program on his computer to send control commands to an Arduino Uno, which adjusts the filter operating frequency.

ARRL The national association for AMATEUR RADIO™
225 Main Street
Newington, CT USA 06111-1494

Taking HF By Storm

The TS-480HX

KENWOOD

Customer Support: (310) 639-4200
Fax: (310) 537-8235

Scan with your phone to download TS-480HX brochure.

www.kenwood.com/usa

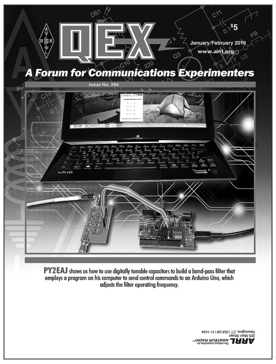
ISO9001 Registered
JVCKENWOOD Corporation

ADS#27315

QEX

About the Cover

Euclides Lourenço Chuma, PY2EAJ, designed a digitally tunable band-pass filter using Peregrine PE64102 digitally tunable capacitor ICs. He controls the operating frequency of the filter with a program on his computer that sends commands to an Arduino Uno board, which then sets the capacitor values.


QEX
January/February 2016
A Forum for Communications Experimenters
Issue No. 294
PY2EAJ shows us how to use digitally tunable capacitors to build a band-pass filter that employs a program on his computer to send control commands to an Arduino Uno, which adjusts the filter operating frequency.
ARRL

QEX (ISSN: 0886-8093) is published bimonthly in January, March, May, July, September, and November by the American Radio Relay League, 225 Main Street, Newington, CT 06111-1494. Periodicals postage paid at Hartford, CT and at additional mailing offices.

POSTMASTER: Send address changes to: QEX, 225 Main St, Newington, CT 06111-1494
Issue No 294

Harold Kramer, WJ1B
Publisher

Larry Wolfgang, WR1B
Editor

Lori Weinberg, KB1EIB
Assistant Editor

Zack Lau, W1VT
Ray Mack, W5IFS
Contributing Editors

Production Department

Steve Ford, WB8IMY
Publications Manager

Michelle Bloom, WB1ENT
Production Supervisor

Sue Fagan, KB1OKW
Graphic Design Supervisor

David Pingree, N1NAS
Senior Technical Illustrator

Brian Washing
Technical Illustrator

Advertising Information Contact:

Janet L. Rocco, W1JLR
Business Services
860-594-0203 – Direct
800-243-7768 – ARRL
860-594-4285 – Fax

Circulation Department

Cathy Stepina, *QEX Circulation*

Offices

225 Main St, Newington, CT 06111-1494 USA
Telephone: 860-594-0200
Fax: 860-594-0259 (24 hour direct line)
e-mail: **qex@arrl.org**

Subscription rate for 6 issues:

In the US: ARRL Member $24, nonmember $36;

US by First Class Mail: ARRL member $37, nonmember $49;

International and Canada by Airmail: ARRL member $31, nonmember $43;

Members are asked to include their membership control number or a label from their QST when applying.

In order to ensure prompt delivery, we ask that you periodically check the address information on your mailing label. If you find any inaccuracies, please contact the Circulation Department immediately. Thank you for your assistance.

In This Issue

Features

Index of Advertisers

ARRL	Cover III	M^2:	19
Down East Microwave Inc:	36	Nemal Electronics International, Inc:	36
DX Engineering:	22	Quicksilver Radio Products	Cover IV
Kenwood Communications:	Cover II	RF Parts:	15, 17
		Tucson Amateur Packet Radio:	36

The American Radio Relay League

The American Radio Relay League, Inc, is a noncommercial association of radio amateurs, organized for the promotion of interest in Amateur Radio communication and experimentation, for the establishment of networks to provide communications in the event of disasters or other emergencies, for the advancement of the radio art and of the public welfare, for the representation of the radio amateur in legislative matters, and for the maintenance of fraternalism and a high standard of conduct.

ARRL is an incorporated association without capital stock chartered under the laws of the state of Connecticut, and is an exempt organization under Section 501(c)(3) of the Internal Revenue Code of 1986. Its affairs are governed by a Board of Directors, whose voting members are elected every three years by the general membership. The officers are elected or appointed by the Directors. The League is noncommercial, and no one who could gain financially from the shaping of its affairs is eligible for membership on its Board.

"Of, by, and for the radio amateur," ARRL numbers within its ranks the vast majority of active amateurs in the nation and has a proud history of achievement as the standard-bearer in amateur affairs.

A *bona fide* interest in Amateur Radio is the only essential qualification of membership; an Amateur Radio license is not a prerequisite, although full voting membership is granted only to licensed amateurs in the US.

Membership inquiries and general correspondence should be addressed to the administrative headquarters:

ARRL
225 Main Street
Newington, CT 06111 USA
Telephone: 860-594-0200
FAX: 860-594-0259 (24-hour direct line)

Officers

President: KAY C. CRAIGIE, N3KN
570 Brush Mountain Rd, Blacksburg, VA 24060

Chief Executive Officer: DAVID SUMNER, K1ZZ

The purpose of *QEX* is to:

1) provide a medium for the exchange of ideas and information among Amateur Radio experimenters,

2) document advanced technical work in the Amateur Radio field, and

3) support efforts to advance the state of the Amateur Radio art.

All correspondence concerning *QEX* should be addressed to the American Radio Relay League, 225 Main Street, Newington, CT 06111 USA. Envelopes containing manuscripts and letters for publication in *QEX* should be marked Editor, *QEX*.

Both theoretical and practical technical articles are welcomed. Manuscripts should be submitted in word-processor format, if possible. We can redraw any figures as long as their content is clear.
Photos should be glossy, color or black-and-white prints of at least the size they are to appear in *QEX* or high-resolution digital images (300 dots per inch or higher at the printed size). Further information for authors can be found on the Web at **www.arrl.org/qex/** or by e-mail to **qex@arrl.org**.

Any opinions expressed in *QEX* are those of the authors, not necessarily those of the Editor or the League. While we strive to ensure all material is technically correct, authors are expected to defend their own assertions. Products mentioned are included for your information only; no endorsement is implied. Readers are cautioned to verify the availability of products before sending money to vendors.

Larry Wolfgang, WR1B

Empirical Outlook

A New Year Awaits Us!

For many of us, a new year means excitement and anticipation of everything that the new year will bring. We make resolutions (maybe even writing some of them down) and plan to keep them throughout the year. (Well, okay, most of us are realistic enough to know we'll be lucky to keep them until February, but isn't every new year an opportunity to try again? Maybe this will be the year!)

For *QEX*, we anticipate six issues filled with the technical content, projects, and theoretical discussions to which we all look forward. I know there are some excellent articles waiting to be published. I also know that some of our readers are planning to write an article about their favorite topic, and submit it to *QEX* in the hopes of seeing it in print some day soon. Whether you are interested in presenting some ideas for a new technical development, reporting on some experimentation you have done, or describing a new project that you want to share, *QEX* looks forward to hearing from you!

Each of our readers and authors have the opportunity to help *QEX* maintain its reputation as the best Amateur Radio technical literature available anywhere. I have often encouraged our readers to submit articles for ARRL publications. We depend on you to share the technical material that you like to read. I have talked with some authors who were reluctant to write for *QEX* because they believed that their article would not be technical enough for this magazine. There has been a perception that while much of the technical content of *QST* is written at a simpler level, with shorter articles that will appeal to neophytes and those with less technical backgrounds, *QEX* articles must be much more advanced, or even complicated. Whenever I have been confronted by a potential author who has that attitude, I have tried to convince them that there is no "technical gap" between articles suitable for publication in *QST* and those suitable for publication in *QEX*.

The ARRL technical editorial staff gives careful consideration to each article that is submitted for possible publication. Each article is reviewed by some of our ARRL Technical Advisors, and they provide feedback about the technical content and the accuracy of the information contained in the article. Then each article is examined to determine how to best make use of the material. Some articles are selected for publication in *QST* (even some articles that were submitted for *QEX*) and some are selected for publication in *QEX* (even some that were submitted for *QST*). The editorial staff may even suggest some articles for publication in *NCJ*, or occasionally even one of our books, such as *The ARRL Handbook* or *The ARRL Antenna Book*. Of course there are also some articles that just don't fit the needs of ARRL publications at that time.

My point is that you should write your article and submit it for possible publication. Let the ARRL editorial staff determine the publication for which it may be best suited. I can assure you, however, that I have never seen an article that was considered too complex or technical for *QST*, but too simple for *QEX*.

I have recently had several discussions with a reader who expressed some concern that the content of *QEX* was becoming too focused on construction projects and articles describing how the author had built the project, often omitting details about how they developed that design. This reader was lamenting "the good old days" when the articles in *QEX* seemed much more focused on purely technical discussions and design considerations, perhaps from an engineering perspective. I'll admit that I like construction projects, and I have enjoyed presenting many projects in the pages of *QEX*. The fact remains, however, that you, our readers drive the content of *QEX* by what you submit for publication. If most of the articles submitted are detailed construction projects, then there will be more of those printed. If more articles with detailed technical discussions are submitted, then there will be more of that type of article to present.

The continued success of *QEX* depends on you, our readers, and our authors. I truly hope *QEX* can continue to be strong technical publication. Unfortunately, I will no longer be guiding *QEX* as its Editor. With the change from 2015 to 2016, I will no longer be a member of the ARRL Headquarters Staff. I have enjoyed 34 ½ years on the staff, first as an Assistant Technical Editor and then as a Senior Assistant Technical Editor. For the last 8 ½ years I have served as *QEX* Editor. I have enjoyed working with all of the authors, and I have also enjoyed meeting many authors and readers in person, at various conferences and conventions. If our paths cross at some future date, I hope you will pause and say hello. Please continue to support *QEX*, and help it remain a strong technical publication.

73,

Larry Wolfgang, WR1B

Euclides Lourenço Chuma, PY2EAJ

Rua Cel Manuel de Moraes 204, Campinas, SP 13073-022, Brazil; **codigocerto@yahoo.com.br**

Digitally Tunable Band-Pass Filter

Computer control tunes this versatile band-pass filter to meet changing requirements.

Band-pass filters have always been important, but new hardware technology has made them even more significant. In environments with large amounts of electromagnetic pollution, this new type of band-pass filter can be very helpful.

To construct the digitally tunable band-pass filter, I used digitally tunable capacitors (DTC). These components have a variable capacitance that can be controlled digitally. The digitally tunable band-pass filter in this article covers a frequency range of 132 MHz to 148 MHz, and can handle up to 26 dBm of RF power. It can be integrated into different projects, including software defined radio (SDR) projects, where the band-pass filters are of fundamental importance to the end result.

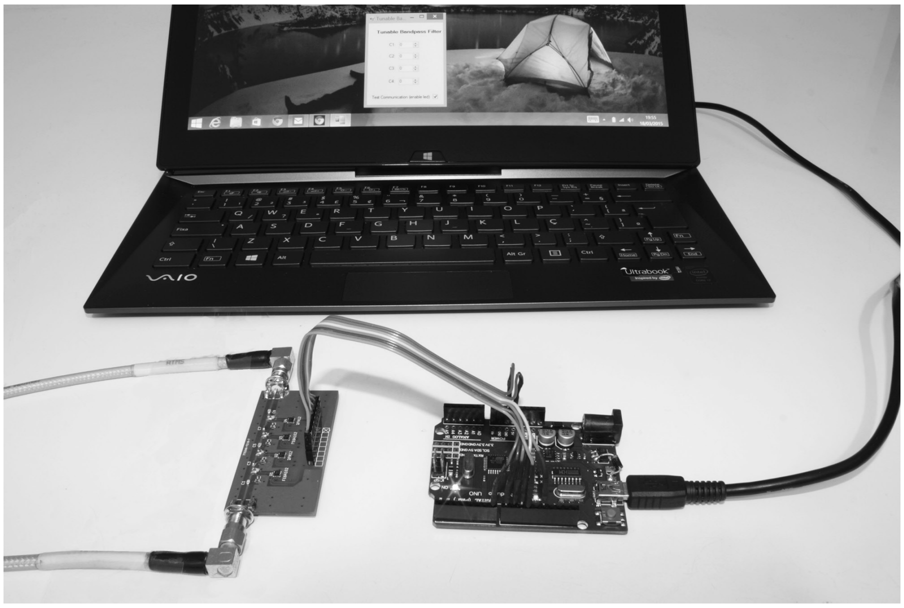
Digitally Tunable Capacitors (DTCs)

The key parts of the band-pass filter described in this article are digitally tunable capacitors (DTCs), which are made up of several high-*Q*-factor metal-insulator-metal (MIM) capacitors, which are digitally controlled FET switches. Figure 1 is a functional block diagram of a DTC.

The DTC used in the design of this project is the Peregrine PE64102, which has a capacitance range of 1.88 pF to 14.0 pF in steps of 391 fF, totaling 32 states that are controlled with 5 bits through a serial peripheral interface (SPI) bus.[1] The Peregrine PE64102 works at 100 MHz to 3 GHz, and can handle up to 26 dBm of RF power.

Design the Filter

Basically, a band-pass filter is used to attenuate all signals having frequencies out-



The Computer, Arduino Uno and band-pass filter circuit board.

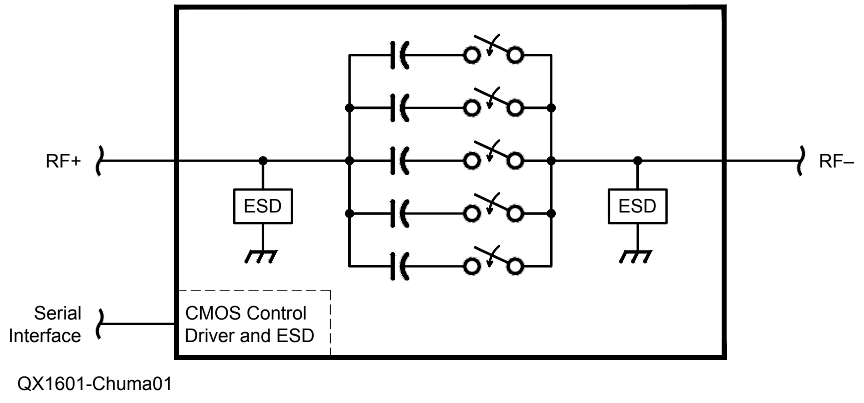


Figure 1 — This is a functional block diagram of the digitally tunable capacitors.

[1]Notes appear on page 8

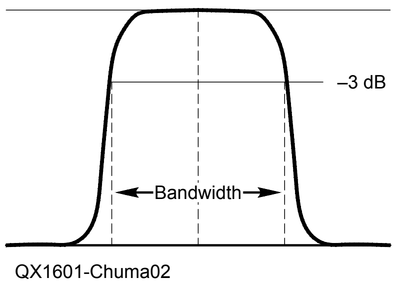


Figure 2 — This is an idealized response curve of a band-pass filter.

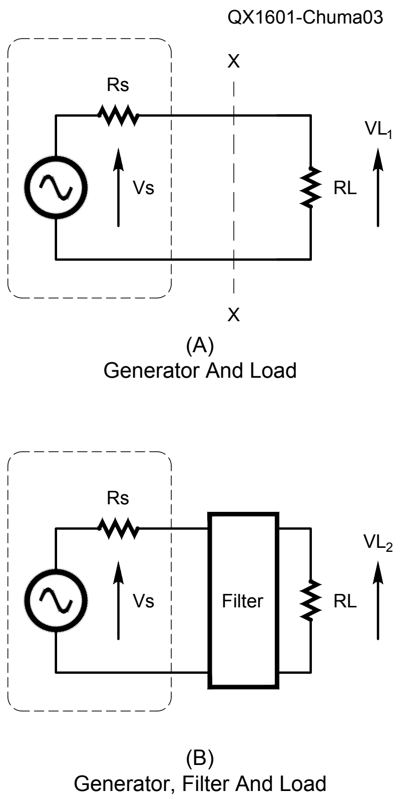
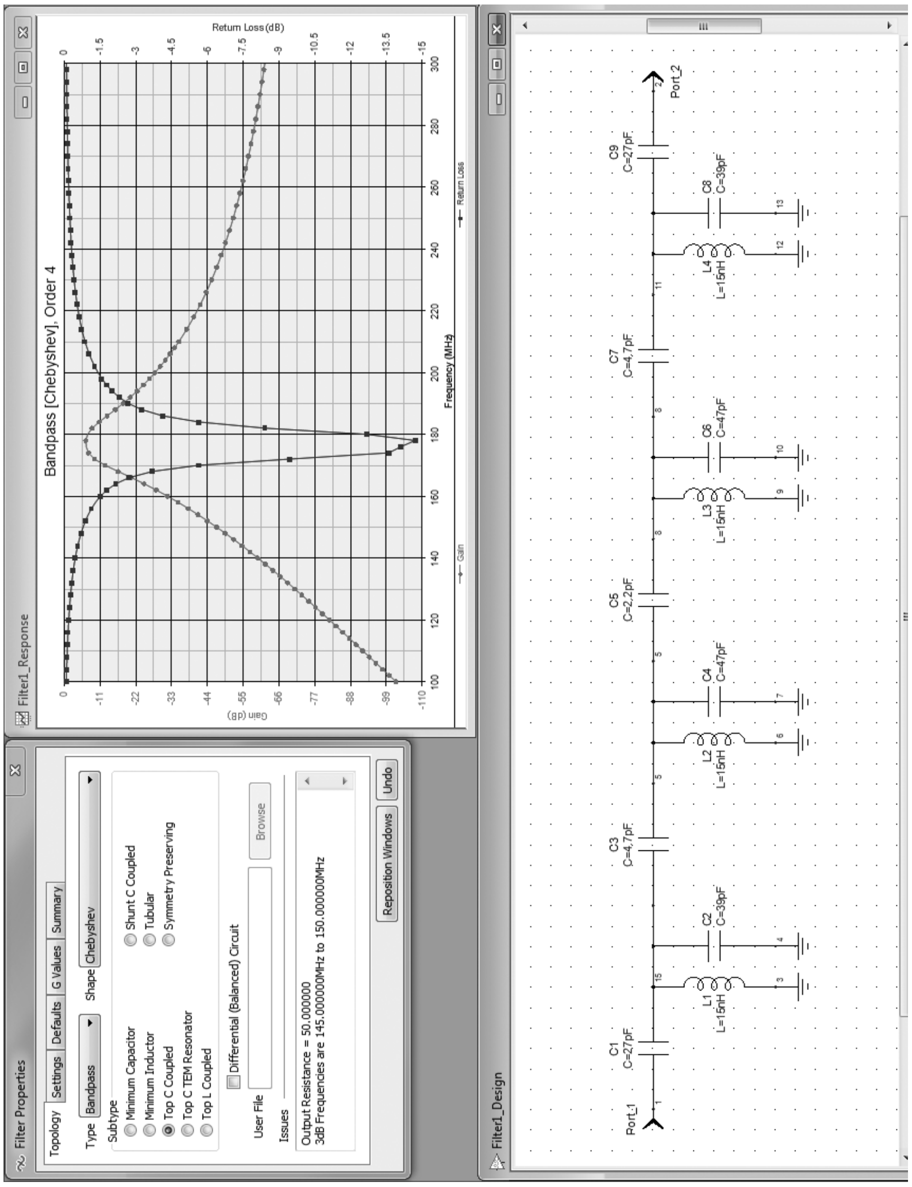


Figure 3 — Here is a block diagram that explains filter insertion loss.



Type: Bandpass
Shape: Chebyshev
Subtype:
- Minimum Capacitor
- Minimum Inductor
- [✓ Top C Coupled]
- Top C TEM Resonator
- Top L Coupled
- Shunt C Coupled
- Tubular
- Symmetry Preserving

Differential (Balanced) Circuit
User File
Issues
Output Resistance = 50.000000
3dB Frequencies are 145.000000MHz to 150.000000MHz

Figure 4 — This is a screen shot of the filter simulation using the Agilent *Genesys 2010* software.

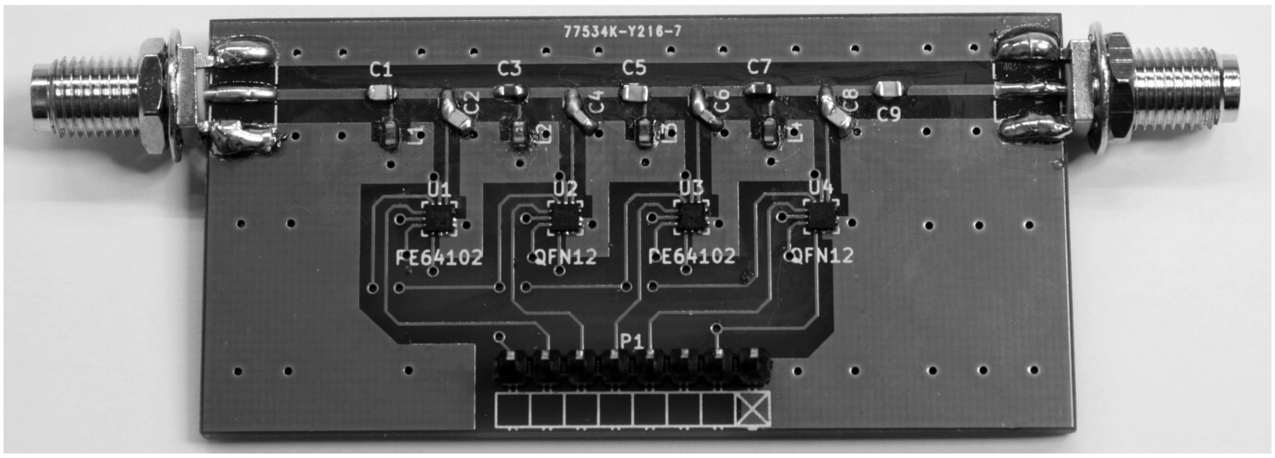


Figure 5 — This photo shows the completed filter circuit board.

side the band of interest. Figure 2 shows the frequency response of a real band-pass filter, where f_c is the center frequency and the bandwidth is the difference between frequencies f_1 and f_2, where the attenuation is 3 dB compared with the magnitude of f_c.

To develop a band-pass filter, we need to know several filter details, such the center frequency, bandwidth, terminating resistance, type of filter response (Butterworth, Chebychev, Elliptical, and so on), and filter order. With these parameters, we can design our filter using simulation software or through manual calculations (hard work).

Several criteria are used to measure the performance of a filter, but the ones used most are the insertion loss and return loss. Insertion loss is the loss of signal power resulting from the insertion of a filter in the transmission line. In Figure 3, if $R_L = R_s$ then V_{L1} is half of V_s. If you add a filter between the generator and the load at position X-X, then any additional loss is the result of the insertion loss of the filter.

For the band-pass filter development of this article I used the *Genesys 2010* software from Agilent. I chose the band-pass filter "Top C Coupled" and Chebyshev type, of order 4 and with 5 MHz of bandwidth. After the first tests, it became clear that there was a downshift of the frequency range, and so I made some changes in component values to achieve the frequency range of 132 MHz to 148 MHz.

The model "Top C Coupled" filter was chosen to use fewer inductors (where Q is critical), and also to come up with a design that uses component values that are available on the market.

Building It

The biggest challenge in building the filter was the size of the digitally tunable capacitors. They are available as a quad flat-pack, no-lead 12-pin package with dimensions of 2 × 2 mm and 0.5 mm pin spacing!

The circuit board design was done in *Kicad* software and the manufacture was made with common characteristics (2-Layer FR4 1.6 mm).

The capacitors used are commonly available, with 5% tolerance, but the ideal would be to use pre-selected NP0 capacitors. The inductors are manufactured by Coilcraft, and have a Q of about 40 at 150 MHz. The inductors also offer a certain challenge to be soldered in place on the circuit board. The finished circuit board is shown in Figure 5. Figure 6 is the schematic diagram of the digitally tunable band-pass filter.

Controls

I adapted an Arduino Uno circuit board

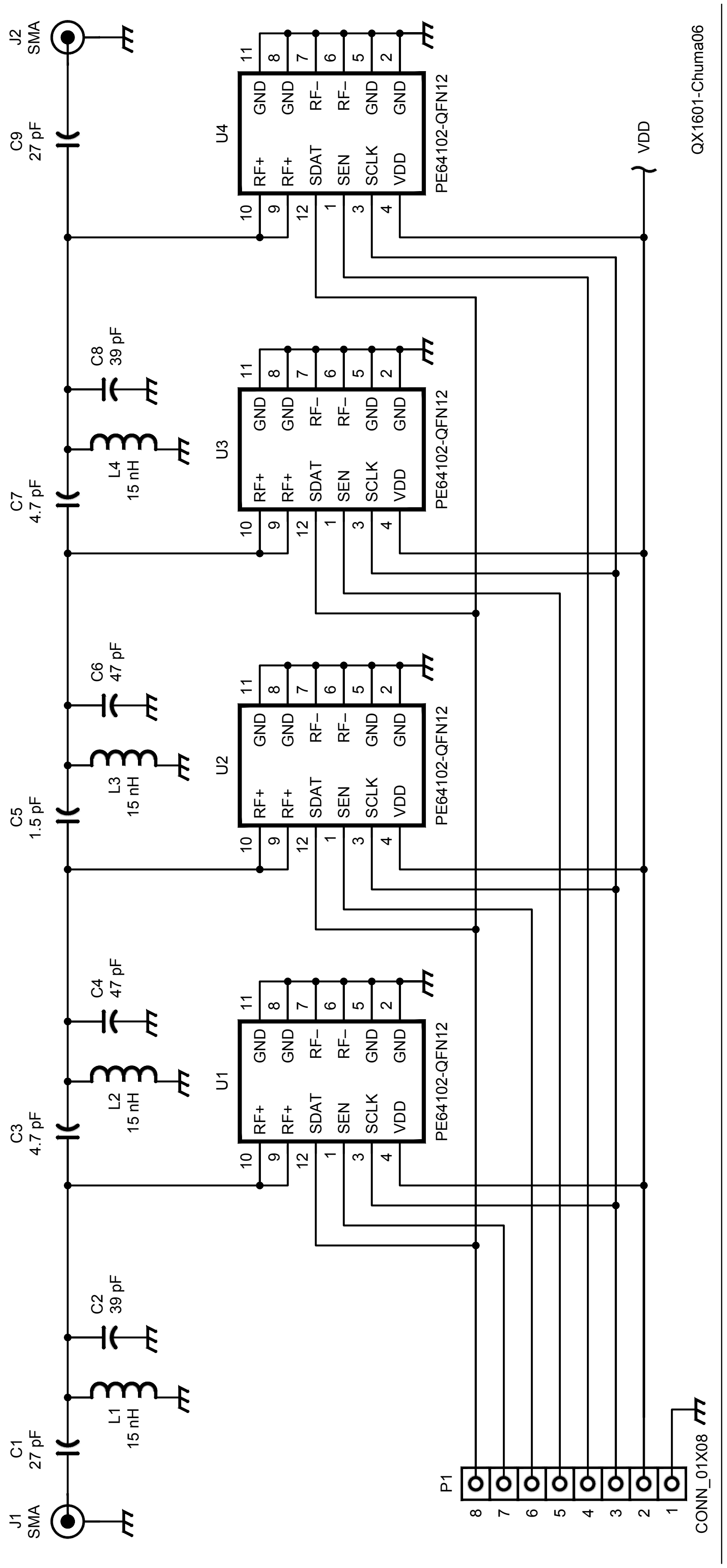


Figure 6 — Here is the schematic diagram of the digitally tunable band-pass filter.

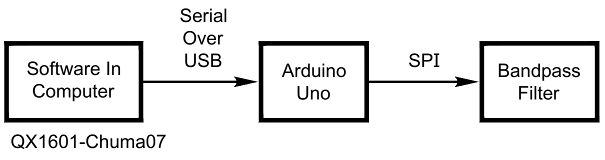
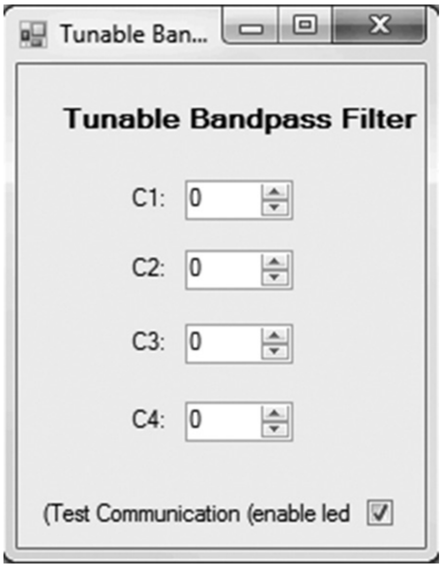


Figure 7 — This block diagram shows the control of the band-pass filter.



Tunable Ban...

Tunable Bandpass Filter

C1: 0

C2: 0

C3: 0

C4: 0

(Test Communication (enable led ☑

Figure 8 — This is a screen shot of the filter control software.

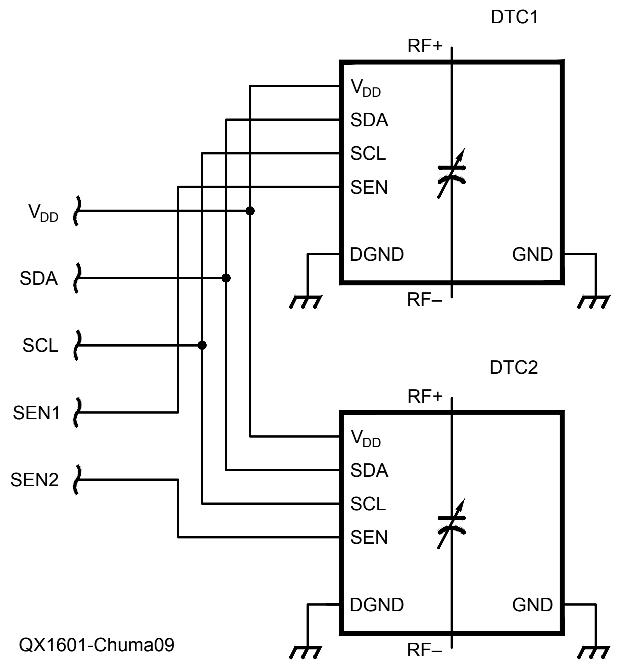


Figure 9 — Here is the serial peripheral interface (SPI) bus sharing to control the digitally tunable capacitors.

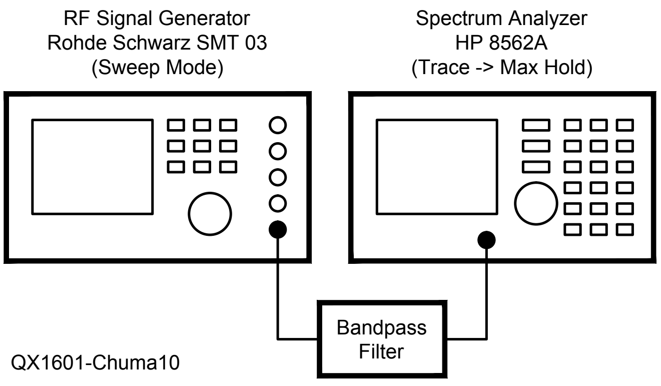


Figure 10 — This diagram shows the test set-up for the filter.

for the control of the digitally tunable capacitors.[2] The Arduino Uno board is controlled by software on a PC that sends commands through a serial-over-USB interface. The lead photo shows the complete set-up, with a laptop computer, the Arduino Uno board and the band-pass filter. Figure 7 is a block diagram of the system.

I adopted this control configuration because it is simple, inexpensive, and allows various filter settings with few hardware changes. The choice of the Arduino Uno was motivated because it is simple to program and has low cost.

The software on the computer was programmed in *C#* with Microsoft *Visual Studio*, using the CmdMessenger library.[3] The software for the Arduino Uno was programmed in Arduino language, which is relatively simple. Figure 8 is a screen shot of the control software for the tunable band-pass filter.

The four PE64102 digitally tunable capacitors use a 3-wire serial 8-bit interface compatible with a serial peripheral interface (SPI) bus, to be controlled by the Arduino Uno board. The digitally tunable capacitor selection is made through the serial enable signal to each capacitor, and then shared on the same bus between all of the capacitors. See the schematic of Figure 9.

The PE64102 digitally tunable capacitors operate with 2.3 V to 3.6 V, and the Arduino Uno board takes 5 V. Thus, I modified the Arduino Uno board to operate with 3.3 V, and thereby the serial peripheral interface bus signals are compatible with these capacitors.[4] There are also other options to convert the serial peripheral interface bus signals of 5 V to 3.3 V.

Results

Although the most appropriate piece of test equipment to adjust the filter would be a vector network analyzer, I did not have one available. Instead, for testing, I used a Rohde & Schwarz SMT 03 RF signal generator with the sweep function enabled and plugged into the filter connector. At the other end was an HP 8562A spectrum analyzer with the "Trace -> Max Hold" function enabled. Figure 10 illustrates these connections.

The results are satisfactory, as can be seen in Figures 11, 12, and 13. For a more detailed analysis a vector network analyzer is needed.

The results point to a downshift of frequency compared to the simulation software. One of the possible causes could be the parasitic capacitance of the circuit board, which was developed with simple software and common materials. Another possible cause is the tolerances of capacitors and inductors. Parasitic capacitances, parasitic resistances, and parasitic inductances of the PE64102

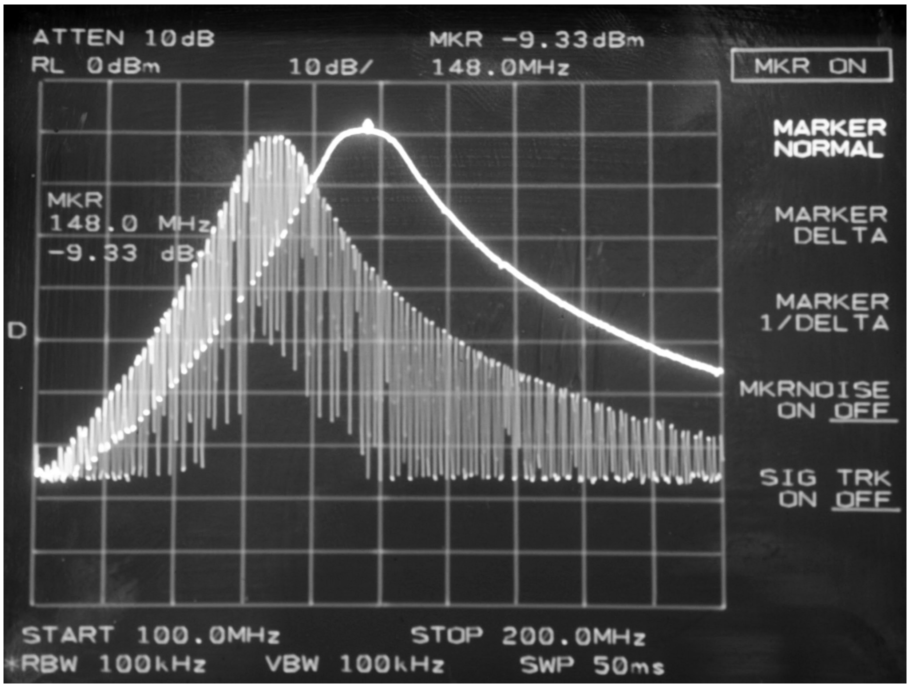


Figure 11 — This spectrum analyzer photo shows the results of the filter test. There are two scans, with center frequencies of 132 MHz and 148 MHz.

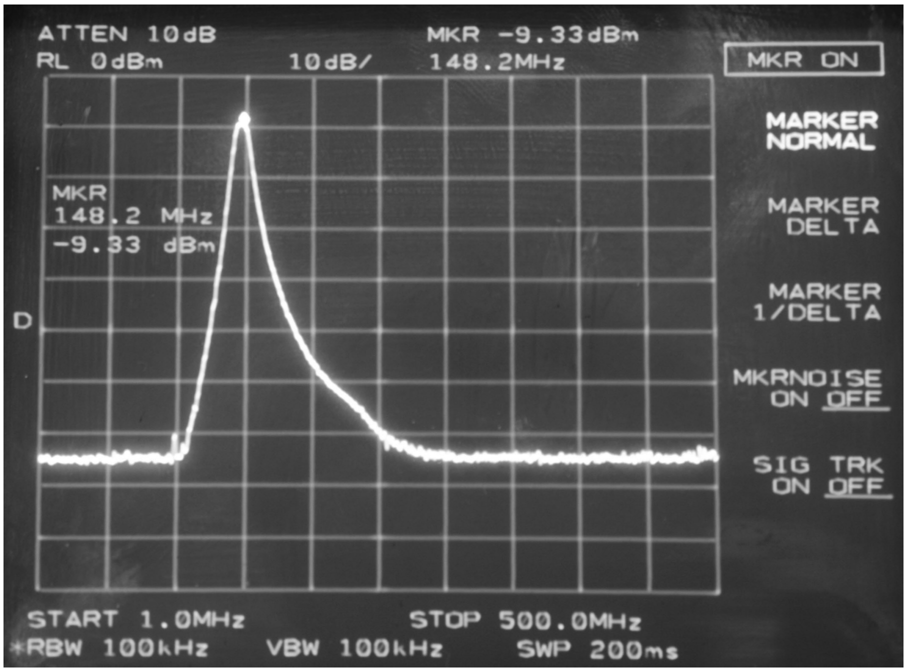


Figure 12 — This spectrum analyzer photo shows a scan of the filter, with the center frequency at 148 MHz.

digitally tunable capacitors may also contribute to this frequency shift. Figure 14 shows a model of the filter circuit, with all of the parasitic components.

Conclusions

The positive points of this project are the linearity of the tuning center frequency, low cost, simplicity and small size. This design is a starting point for more complex projects, such as a set of digitally controlled filters for multiple bands, or even to form part of an SDR design with an adjustable band-pass filter that is controlled digitally by the same SDR software. Then, when the SDR frequency is changed, the software automatically changes the frequency of the band-pass filter.

Euclides Lourenço Chuma, PY2EAJ, earned a degree in mathematics from Unicamp, and a graduate degree in software engineering. Currently, he is working on a graduate degree in network and telecommunications systems in the INATEL. He has been a licensed Amateur Radio operator since 2008, and lives in Brazil. He works in software development, and is interested in everything about RF. He is also interested in antennas, microwaves, SDR and cognitive radio.

Notes

[1]You can find the Peregrine PE64102 datasheet at: **www.psemi.com/pdf/datasheets/pe64102ds.pdf**.

[2]For details about the Arduino Uno, go to: **arduino.cc/en/main/arduinoBoardUno**

[3]To learn more about the Arduino Uno CmdMessenger and to download the files, go to: **playground.arduino.cc/Code/CmdMessenger**.

[4]Details about converting the Arduino Uno to operate from 3.3 V are available at: https://learn.adafruit.com/arduino-tips-tricks-and-techniques/3-3v-conversion. Software and circuit board files are available at: **www.chumalab.com.br/digitally-tunable-bandpass-filter/**

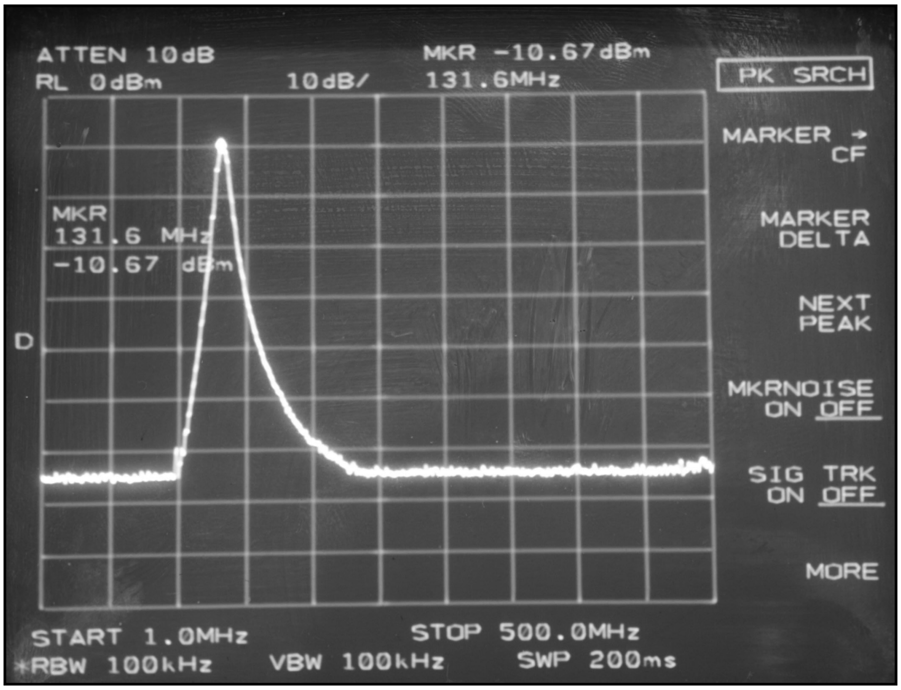


Figure 13 — This spectrum analyzer photo shows a scan of the filter, with the center frequency at 132 MHz.

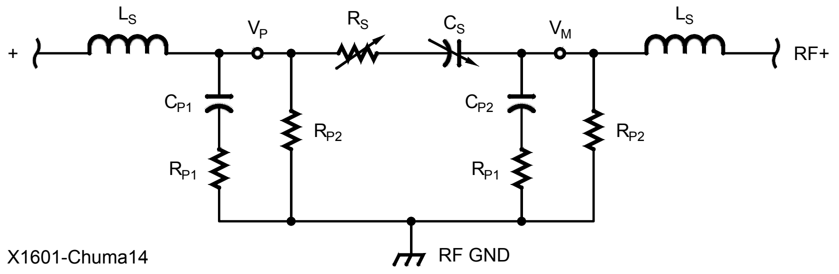


Figure 14 — This is a model of the equivalent circuit of the PE64102 digitally tunable capacitor.

Letters to the Editor

A Frequency Standard for Today's WWVB (Nov/Dec 2015)

Hi Larry,

After the Nov/Dec 2015 issue went to press, I discovered several errors in the text and drawings. In several places, WWVB signal levels expressed in μV/m were printed as mV/m. These occurred on pages 14 (caption to Figure 3), 16 (bottom of middle text column), and 29 (*e* = local field strength (mV/m)).

Several of the schematic diagrams also had some errors. In Figure 7, there is a missing connection between U2 pin 7 and the right side of R10. On Figure 12b, U38 is mislabeled. U38 should be an LMC6484N. Figure 13 had several wiring errors around U14B and U15A. Also there is no connection between U17 pin 7 and the BCD Switch. That pin should connect to ground. The Output Frequency Selector should have listed signals at 25 kHz and 10 kHz. On Figure 16, op-amps U1D and U6D should be labeled as LM837N devices.

— 73, John Magliacane, KD2BD, 1320 Willow Dr, Sea Girt, NJ 08750; **kd2bd@amsat.org**

Hi John,

My apologies for all of those errors. We continue to be plagued by Greek characters in the Symbol font changing back to normal text at times, and no one noticed those Greek mu characters (μ) were replaced by the normal letter m.

I don't believe we have ever had that many errors on schematic diagrams in a single article. That is embarrassing! I learned later that the *QEX* contract graphics artist was having some personal issues, and he did not do his normally careful drawing work on your article. We have corrected all of the errors that you have called to my attention. Your article is the Sample Article for the Nov/Dec 2015 issue. The fully corrected article is available for viewing and download on the "This Month in *QEX*" page on the ARRL website. Readers can go to **www.arrl.org/files/file/QEX_Next_Issue/2015/Nov-Dec_2015/Magliacane.pdf**.

— 73, Larry Wolfgang, WR1B, QEX Editor; **lwolfgang@arrl.org**

David L. Hershberger, W9GR

10373 Pine Flat Way, Nevada City, CA 95959: **w9gr@arrl.net**

External Processing for Controlled Envelope Single Sideband

It is now possible to separate the CESSB processing from the transmitter.

In my Nov/Dec 2014 *QEX* article on controlled envelope single sideband (CESSB), I stated that generation of the CESSB signal is best integrated into the SSB modulator of a radio, rather than being done in an external box.[1] It is possible to separate CESSB generation from a radio, however, if the radio SSB modulator is designed with this in mind.

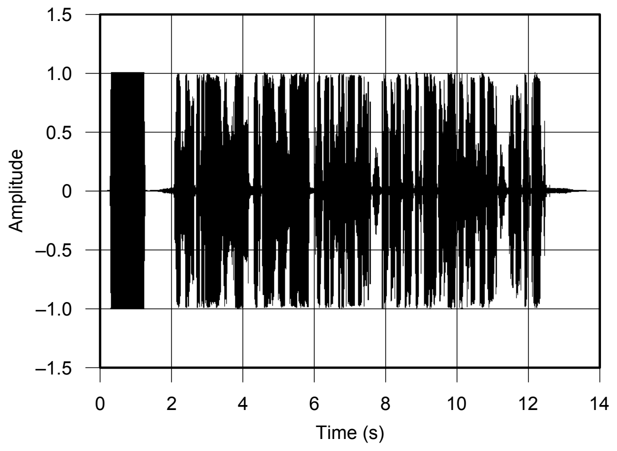
The SSB modulator must be linear phase, and must have a bandwidth sufficient to pass the CESSB spectrum, including its spectral skirts. If an otherwise conventional SSB modulator meets these requirements, then the peak control obtained by the CESSB process will be preserved.

This will make it possible to use external processing to create CESSB. The radio may be used for conventional SSB if an external CESSB processor is not available.

The envelope control problem with single sideband is that limiting audio peaks does not accurately limit SSB envelope peaks. The envelope of an SSB signal is basically the vector magnitude of the modulating audio signal plus its Hilbert Transform. The Hilbert Transform is an audio phase shift of 90° for all frequencies within its bandwidth. The Hilbert Transform overshoots, making RF envelope amplitude control difficult.

CESSB is a way of controlling the inevitable RF envelope overshoots caused by the Hilbert Transform. These Hilbert Transform overshoots occur regardless of the method used to generate SSB. A phasing method SSB modulator produces a Hilbert Transform directly, by means of audio phase shift networks. Filter and Weaver method SSB modulators produce the Hilbert Transform indirectly.

[1]Notes appear on page 12



QX1601-Hershberger01

Figure 1 — An externally processed CESSB signal, reproduced by a linear phase Hilbert Transform SSB modulator.

If the envelope overshoots are not reduced, then ALC or manual transmit gain control will reduce the SSB signal amplitude, such that there is no flat-topping. This reduces average transmitted power.

Conversely, if the Hilbert Transform-induced envelope peaks are reduced or eliminated, then the average transmitted power of an SSB signal can be significantly increased. A 2.5 dB increase in average transmitted power is typical, compared with advanced look-ahead ALC systems.

Discussion

The intermediate output of the CESSB process is a pair of audio baseband signals. These are often known as "I" and "Q" signals, for in-phase and quadrature. If the I and Q audio signals are applied to a pair of mixers driven with quadrature RF, then the sum of the two mixer outputs will be SSB.

Another characteristic of the I and Q signals is that they are interrelated by a Hilbert Transform, or a negative Hilbert Transform. In other words, the audio signals are 90° out of phase between I and Q at all frequencies. In that regard, there is redundancy in I and Q.

One way to separate the CESSB process would be to pass the two baseband I and Q audio signals to a radio. It would be important to maintain accurate amplitude and phase matching for the two audio signals. It is not necessary to pass both audio signals into an SSB transmitter, however.

Because the two I and Q outputs of the CESSB system contain redundancy, you can throw one of them away and then regenerate it if necessary. The remaining signal has a special characteristic. The vector magnitude (or modulus) of itself plus its Hilbert Transform, is accurately amplitude limited. That vector magnitude function is proportional to the RF envelope amplitude of the SSB signal.

$$e(t) = \sqrt{a^2(t) + H^2\left[a(t)\right]} \qquad \text{[Eq 1]}$$

where:
$e(t)$ is the envelope signal
$a(t)$ is the input audio signal
$H[a(t)]$ is the Hilbert Transform of the input audio signal.

What Equation 1 suggests is that we could discard either the I or the Q signal, and pass just one audio baseband signal as $a(t)$ from the external CESSB processor to the radio. The radio could then regenerate the missing signal with a Hilbert Transform (either directly or indirectly). If this is done with linear phase and flat amplitude response, then the regeneration of the discarded signal will be perfect.

For this to work, the radio must have a linear phase response in its SSB modulator. That means flat time delay versus frequency. Also, the frequency response of the SSB modulator must be equal to or greater than the skirt bandwidth of the CESSB I and Q signals.

So, if the CESSB signal has a response of 300 to 3000 Hz, with descending filter skirts extending to 150 Hz at the low end and 3150 Hz on the high end, then the SSB modulator in the radio should have flat amplitude and linear phase from 150 to 3150 Hz. As long as those conditions are met, the radio will transmit accurately controlled envelope peaks using an external CESSB processor.

Unfortunately, most of the analog SSB transmitters in use today do not have linear phase response. A conventional radio with a crystal or mechanical filter for SSB generation might be wide enough, but it will have group delay peaks near the band edges. On the other hand, some SSB transmitters using DSP may very well have linear phase response. Those radios, if they exist, could be converted to CESSB operation with an external CESSB processor.

Simulations

GNU *Octave* is an excellent simulation and signal processing tool.[2] I have written some GNU *Octave* code that simulates the external CESSB system. My GNU Octave code is available for download from the ARRL *QEX* files web page.[3] The *Octave* script reads in an audio WAV file, which has been accurately amplitude limited. CESSB processing is done first. Next, one of the two baseband audio signals produced by the CESSB process is discarded. (Actually, the script uses a linear combination of I and Q to produce a single output signal. Any linear combination will work, such as I + Q, I – Q, 0.5 × I – 0.866 × Q, and other combinations). The remaining CESSB audio baseband signal is applied to the following modulators:

1) A linear phase filter type SSB modulator.

2) A linear phase Hilbert Transform SSB modulator.

3) A linear phase Weaver method SSB modulator.

Each of these modulators produces an upper sideband signal at 12 kHz. The sampling rate for all signals in the *Octave* code is 48 kHz.

The *Octave* code inserts a shaped 1 kHz tone, one second long, at the beginning of the speech audio. The purpose of the tone is to create an amplitude reference at the PEP limit of the transmitter power amplifier. A single tone does not create overshoot in any SSB modulator. (Simultaneous multiple frequencies are required to produce overshoot.) Note that the amplitude of the tone is a normalized 1.0 in each of the simulations that follow. If CESSB is accurately preserved, then the amplitude of the speech will not exceed 1.0 either.

All of these modulators reproduce the CESSB signal accurately, with tight envelope peak control. As a result, Figures 1, 2, and 3 look almost identical, even though different SSB modulation methods were used to create them.

SSB Modulators that *Do Not* Preserve CESSB

Next the same audio signal is applied to some inappropriate SSB modulators:

1) A nonlinear phase filter type SSB modulator, using a crystal or mechanical filter (such as a Heathkit SB-102, Collins KWM-2, and similar transceivers).

2) A phasing type SSB modulator (such as the vintage Hallicrafters HT-37 transmitter).

These SSB modulators, typical of analog SSB transmitters, introduce linear distortions to the CESSB audio baseband, and they overshoot. Accurate envelope peak control is lost.

The phasing-type modulator simulation

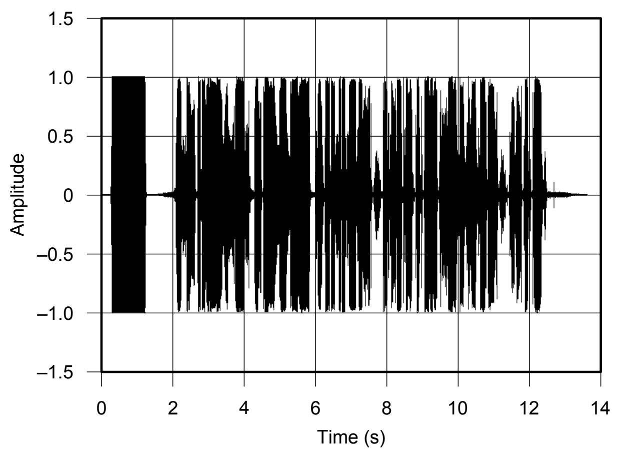


Figure 2 — An externally processed CESSB signal, reproduced by a linear phase bandpass filter SSB modulator.

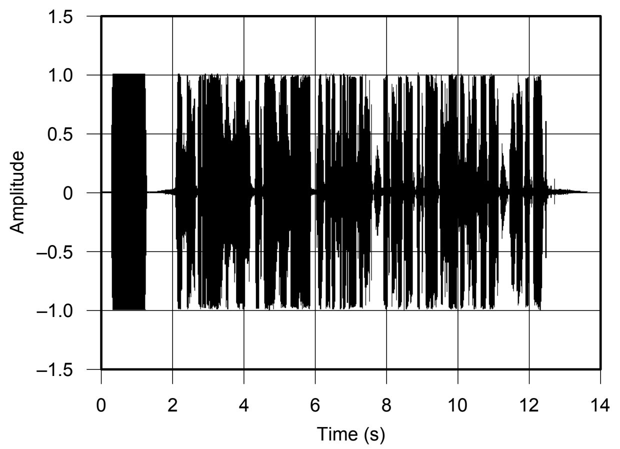


Figure 3 — An externally processed CESSB signal, reproduced by a linear phase Weaver SSB modulator.

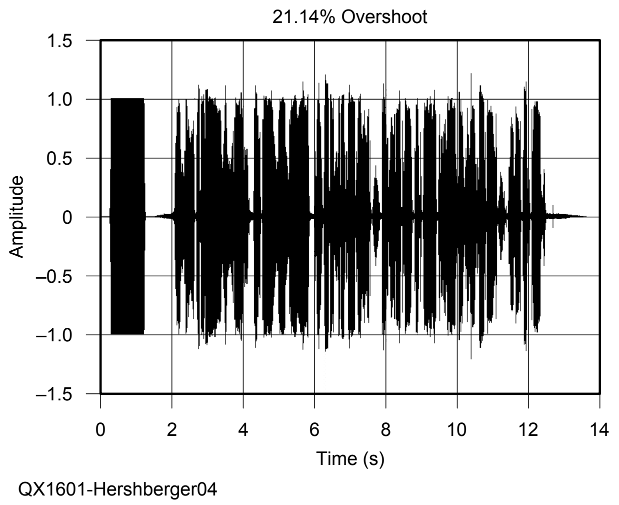


Figure 4 — An externally processed CESSB signal, reproduced by a nonlinear phase filter SSB modulator.

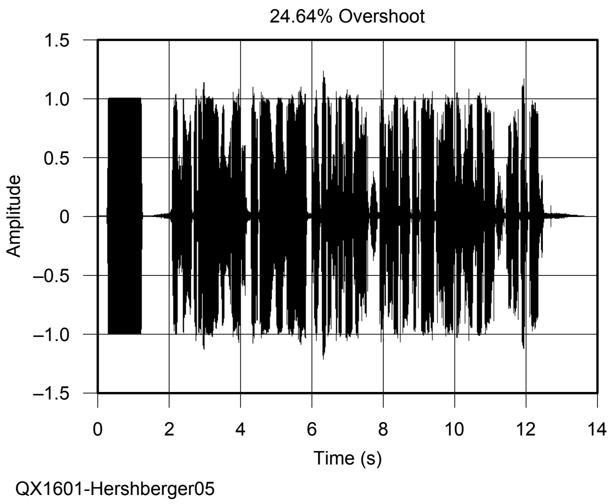


Figure 5 — An externally processed CESSB signal, reproduced by a nonlinear phase phasing method SSB modulator.

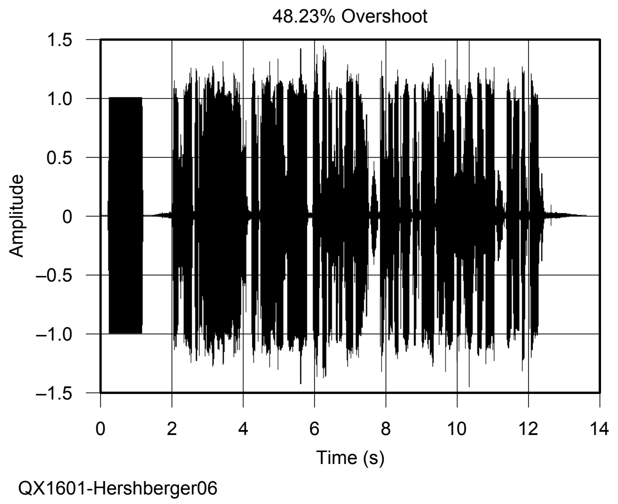


Figure 6 — A peak limited audio signal (not CESSB) applied to a nonlinear phase filter SSB modulator.

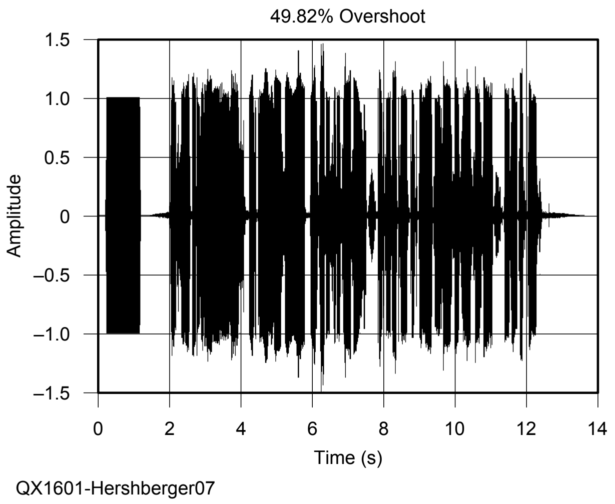


Figure 7 — A peak limited audio signal (not CESSB) applied to a nonlinear phase, phase-difference network SSB modulator.

uses the coefficient set II given by Theodor Prosch, DL8PT, in Table 1 of his Sep/Oct 2012 *QEX* article.[4]

A Hilbert Transform filter, referred to a compensating delay line, has a dϕ/dω characteristic (phase slope) of zero. The phase shift remains at 90° for all frequencies. So, the group delay of a Hilbert Transform is also zero when referred to a compensating delay line. The compensating delay and the Hilbert Transform filter constitute a pair of phase difference networks. Their phase difference is 90° for all frequencies for which the Hilbert Transform filter is designed. Yet, there is no time delay variation versus frequency for either path.

But traditional analog or digital IIR allpass filter phase difference networks do have time delay variations versus frequency and that is what makes a "phasing" type SSB modulator unsuitable for CESSB. The allpass network pair has the following phase shifts:

$\Phi(\omega) + \pi / 2$, and $\Phi(\omega)$

So, it is the $\Phi(\omega)$ phase function that introduces phase distortion and causes overshoot in a phasing-type SSB modulator. Theodor (DL8PT) Prosch's Figure 4 shows the $\Phi(\omega)$ phase function. (See Note 4.) In a true Hilbert Transform modulator, the $\Phi(\omega)$ function is zero, however, a Hilbert Transform modulator requires more computation than a phase-difference network "phasing" type SSB modulator.

The minimum-phase, elliptic type bandpass filter does not work for CESSB because of its group delay variations. The same is true for the phase-difference network SSB modulator. It also has group delay variations.

Using CESSB Processing With Older Analog Radios

While the examples of Figure 4 and Figure 5 show some overshoot when used with CESSB-processed input audio, the overshoot is considerably worse with ordinary peak-limited audio. The same nonlinear phase elliptic filter SSB modulator, when driven from the peak limited audio (no CESSB audio processing) produces the RF envelope shown in Figure 6.

With CESSB audio processing, overshoot is 24.64% instead of 48.23%. Compare Figure 6 to Figure 4. So, even though there is overshoot, there is still some advantage obtained by using a CESSB processor in front of a conventional nonlinear phase filter-type SSB transmitter. With this example, RF power output would be about 1.5 dB greater.

Now let's look at the nonlinear phase, phase-difference network modulator. With conventionally processed audio instead of CESSB audio, Figure 7 shows the RF envelope.

Again, CESSB processing reduces the overshoot from 49.82% to 24.64%. Compare Figure 7 to Figure 5. So even though older nonlinear phase transmitters do not produce true CESSB output from a CESSB audio input, they do benefit from CESSB processing.

Phase equalization (in DSP) of the particular crystal filter, mechanical filter, or phase difference network could certainly reduce the overshoot of these older types of SSB modulators.

Is Your Rig "CESSB-Ready?"

If your rig is a FlexRadio 6000 series, it already has CESSB built-in.

If your transmitter is older or nonlinear phase, it can probably partially benefit from CESSB audio processing.

If you have a modern DSP based transmitter, it might already be fully "CESSB-ready." To find out, you just need to connect a CESSB processor to its audio input and then look at the RF envelope on an oscilloscope.

As of this writing, there are no external CESSB processors available in hardware, but there is still a way to test your rig. The WAV files used to generate the figures in this article are available from the ARRL *QEX* files website. (See Note 3.) All you have to do is play the WAV file (CESSB-ready-test-audio.wav) into your rig and look at the RF envelope coming out. Here are some suggestions:

1) Turn off any equalizers, audio compressors, or other audio processors.

2) If possible, turn off ALC.

3) Run the transmitter power down to about 25% of normal by reducing audio (mic) gain, so you can see any overshoots.

4) If your transmitter has adjustable transmit bandwidth, increase it to about 3.5 kHz or more.

5) Use a dummy load! The audio test files contain my call sign, and you wouldn't want to misidentify your station!

The WAV file contains the reference tone as a maximum PEP reference. If all of the speech peaks stay at or below the reference tone amplitude and look like Figures 1 to 3, congratulations, your rig is CESSB-ready! If the voice peaks visibly exceed the reference tone, and look like Figures 4 through 7, then your rig is not CESSB-ready, but it still may benefit from the use of a CESSB audio processor.

You may also wish to test with the peak-limited-audio.wav file. This file does *not* contain CESSB processing. It only contains simple audio peak limiting. This file will cause SSB modulator overshoot.

The file externalcessb.m is the GNU *Octave* script. Externalcessbmc.m is an edited script that is compatible with Matlab®. Both scripts will create many plots of SSB envelopes, spectra, and filter characteristics.

Conclusions

Although the most convenient way to generate CESSB may be to build it into each radio, CESSB processing can be done with an external box, and radio manufacturers could make radios that are "CESSB-ready." If you just plug in a mic, you don't get CESSB. You get plain old SSB. If you have an external CESSB audio processor, however, then you will get CESSB from a radio that is "CESSB-Ready." Some of the modern DSP rigs might already be "CESSB-ready." Many older analog SSB modulators are not going to preserve CESSB, since they are not linear phase.

If radios that are "CESSB-ready" are made, along with external CESSB processors, then hams will have the option to "mix and match" processors and transmitters. As speech processing algorithms improve, the external CESSB processor can be replaced or upgraded, and the same radio can continue to be used.

The CESSB processor-to-radio interface is a single audio signal. The audio signal path needs to be flat amplitude and linear phase. The SSB modulator also needs to be flat amplitude and linear phase.

Although nonlinear phase transmitters cannot fully preserve the CESSB signal, they do obtain a partial benefit from external CESSB processing.

Dave Hershberger, W9GR, was first licensed in 1965 at age 14 as WN9QCH. He is an ARRL Life Member. Dave holds a bachelor's degree in math from Goshen College and bachelor's and master's degrees in electrical engineering from the University of Illinois. He has been awarded 19 US patents. Dave is Senior Scientist at Continental Electronics. His recent projects include two ATSC digital television broadcast exciters with adaptive linear and nonlinear precorrection, a DSP based FM/HD Radio® exciter with adaptive precorrection, new high power uplink transmitters for the JPL/NASA Deep Space Network, and a 2.4 megawatt VLF transmitter system.

Notes

[1]David L. Hershberger, W9GR, "Controlled Envelope Single Sideband," *QEX*, Nov/Dec 2014, pp 3 – 13. You can download a copy of this article at: **www.arrl.org/files/file/QEX_Next_Issue/2014/Nov-Dec_2014/Hershberger_QEX_11_14.pdf**

[2]There is more information about GNU *Octave* on the *Octave* home page at **www.gnu.org/software/octave**. You can also download the latest version of GNU *Octave* from that website.

[3]The GNU *Octave* files and WAV files are available for download from the ARRL *QEX* files website. Go to **www.arrl.org/qexfiles** and look for the file **1x16_Hershberger.zip.**

[4]Theodore A Prosch, DL8PT, "A Minimalist Approximation of the Hilbert Transform," *QEX*, Sep/Oct 2012, pp 25 – 31.

Joseph J. Roby, Jr, KØJJR

2129 Bel Air Ave, Duluth, MN 55803: **k0jjr@arrl.net**

Introducing AACTOR: A New Digital Mode

The combination of Adaptive Arithmetic Coding and a Modified Version of Radioteletype Results in a Fast Digital Mode.

Radioteletype (RTTY), can be made faster at any baud by combining a data compression and decompression technique called adaptive arithmetic coding (AAC) with a modified version of RTTY called RTTY-AAC.[1, 2, 3, 4, 5]The increase in speed is achieved by using AAC to significantly reduce the number of bits in the message to be sent and received, which, in turn, significantly reduces the number of RTTY Mark and Space tones to be sent and received. This new digital mode is called ***A**daptive **A**rithmetic **C**oding **T**eleprinting **O**ver **R**adio*, AACTOR (pronounced `ak-tor).

In this article I will first explain how AAC compression and decompression work. Then, I will review the basics of RTTY and how RTTY can be modified into RTTY-AAC. I will then present data comparing the speed of RTTY with the speed of AACTOR. I will conclude with a discussion of some practical considerations.

AAC Compression

As the word "arithmetic" in the name "adaptive arithmetic coding" implies, AAC must have something to do with numbers. In fact, AAC compresses a message into a single numeric fraction f, as expressed in Equation 1.

$$0.0 \leq f < 1.0 \qquad \text{[Eq 1]}$$

The process may be viewed as the iterative subdividing of a number line, with the subdivision proportional at each iteration to the number of times each symbol appears in the message up to that point. After the last symbol is processed, f can be any fraction satisfying Equation 2.

$$\textit{low boundary of last subdivision} \leq f < \textit{high boundary of last subdivision} \qquad \text{[Eq 2]}$$

In other words, any f satisfying Equation 2 can be decompressed uniquely by AAC to recreate the original message.

The fraction f must be expressed in such a way that it can be modulated and demodulated with RTTY binary protocols, which are based on Mark and Space audio tones (explained further later). Expressing f with decimal notation will not work. Instead, f must be expressed with binary notation, which, just like decimal notation, can express any fraction.

In binary notation, the first digit to the right of the decimal point represents one-half, the second one-fourth, the third one-eighth, and so on. Thus, 0.1111111111… equals ½+¼+⅛+…, which sums to 1.0. Any fraction can be expressed this way. For example, the decimal fraction 0.375 is 0.011 in binary notation, and the decimal fraction 0.1 is 0.00011001100110011… in binary notation. In binary notation, Equation 1 becomes Equation 3.

$$0.0 \leq f < 0.1111111111\ldots \qquad \text{[Eq 3]}$$

By using binary notation, and by disregarding the 0 to the left of the decimal point as well as the decimal point itself, f simply becomes a stream of binary ones and zeroes. When a fixed-length message is compressed with AAC, the stream will also have a fixed length. Using binary notation and the integer technique explained later, there is no theoretical limit to the precision of f. As measured by bit count, f will always be shorter than the original message, as measured by its RTTY bit count.

To create f with binary notation, first identify the symbol set needed to compose the desired messages. The symbols may be any combination of ASCII characters, such as letters (upper case, lower case, or both), digits, special characters (for example, punctuation), and control characters (for example, carriage return). The symbol set should be as small as possible, however, and should not include any symbols not needed to compose the desired messages. Experimentation indicated that the smaller the symbol set needed to compose the desired messages, the shorter f will be, as measured by bit count.

The collection of symbols, totaling $\boldsymbol{n}$ symbols, will comprise the symbol set $\boldsymbol{S}$. Each symbol within $\boldsymbol{S}$ is identified as $\boldsymbol{s_i}$, as defined by Equation 4.

$$0 \leq \boldsymbol{i} < \boldsymbol{n} \qquad \text{[Eq 4]}$$

The $\boldsymbol{s_i}$ may be in any order, but once ordered, the ordering must remain fixed. Both the compressor and the decompressor must use the same $\boldsymbol{S}$, with the $\boldsymbol{s_i}$ in the same order.

For AACTOR, $\boldsymbol{S}$ contains a collection of $\boldsymbol{n} = 42$ symbols $\boldsymbol{s_i}$ a specific order, which happens to be, but need not be, ASCII order. See Figure 1.

With this $\boldsymbol{S}$, any RTTY-like message may be composed. A slash is included because

[1]Notes appear on page 19

s_0	End of Transmission [EOT]
s_1	Line Feed [LF]
s_2	Carriage Return [CR]
s_3	Space [SP]
s_4	Slash "/"
s_5 to s_{14}	Digits 0 to 9
s_{15}	Question Mark "?"
s_{16} to s_{41}	Upper Case Letters A to Z

Figure 1 — These are the 42 symbols, s_i, contained in the AACTOR symbol set, S.

$$m_0 = 1$$
$$m_1 = 2$$
$$m_2 = 3$$
$$\bullet \quad \bullet$$
$$\bullet \quad \bullet$$
$$\bullet \quad \bullet$$
$$m_{41} = 42$$

Figure 2 — For AACTOR, M is a set of 42 counters, m_i, as shown here.

many Amateur Radio call signs include a slash. The end of text [EOT] character is included so that the AAC decompressor will know when it has reached the end of the message, as explained later. Some punctuation is excluded from S, but unambiguous messages still can be composed. For example, to indicate a new sentence, a [CR] can be used in lieu of a period.

The second step in creating f is to define a way to track how often the s_i appear in the message being compressed. A "model" M handles this task. M contains a collection of n cumulative counters m_i satisfying Equation 4. Each m_i will contain the cumulative count of all s from and including s_0 and up to and including s_i. In other words, if c_i is the count for the number of times s_i appears in the message at any given moment during the compressing of the message, then Equation 5 applies.

$$m_i = \sum_{j=0}^{j=1} c_j \qquad \text{[Eq 5]}$$

The m_i must be in the same order as the s_i, so that the cumulative count for any m_i is always paired with the correct s_i. Like the s_i, the ordering of m_i must remain fixed in both the compressor and the decompressor.

The high bound (hi_i) and low bound (lo_i) for each m_i as they relate to the overall bounds of M (0.0 to 0.1111111111…) are given by Equations 6, 7, and 8.

$$hi_i = m_i \div m_{n-1} \qquad \text{[Eq 6]}$$

$$lo_0 = 0 \qquad \text{[Eq 7]}$$

$$lo_i = hi_{i-1} = m_{i-1} \div m_{n-1} \qquad \textit{for } 0 < i < n \qquad \text{[Eq 8]}$$

When the compressor initializes, it must act like it knows nothing about the message to be compressed. All the compressor can assume is that at least one s_i will appear at least once in the message. Because the compressor must assume it does not know which s_i it will be, the compressor initially must assume that all s_i will appear in the message at least once, even if that assumption later turns out to be incorrect as to some or even most of the s_i. Thus, when the compressor initializes, each m_i will be given by Equation 9.

$$m_i = i + 1 \qquad \text{[Eq 9]}$$

For AACTOR, this means that after initialization, M will be a collection of $n = 42$ cumulative counters m_i as shown in Figure 2.

During compression, M models the message being compressed by updating (adapting) itself as it encounters each symbol in the message. The word "adaptive" appears in the name "adaptive arithmetic coding"

```
append [EOT] to message to be compressed
$r_{lo} = 0.0$
$r_{hi} = 0.1111111111\ldots$
do
        get next symbol
        $r = r_{hi} - r_{lo} + 1$
        find symbol in S and get its index i
        $r_{hi} = r_{lo} + (r \cdot hi_i) - 1$
        $r_{lo} = r_{lo} + (r \cdot lo_i)$
        do
                if $r_{hi}$ and $r_{lo}$ MSBs match
                        process the MSBs (explained in text)
                        process any pending underflow bits (explained in text)
                else if an underflow condition threatens in $r_{hi}$ and $r_{lo}$
                        accumulate and remove underflow bits (explained in text)
                else
                        update (adapt) M based on i
                        exit inner loop
                end if
        loop
        exit outer loop if no more symbols in message to be compressed
loop
flush $r_{lo}$ (explained in text)
```

Figure 3 — This is the Pseudo code for the AACTOR compression loop.

for this reason.[6] For example, if the first symbol encountered in the message being compressed is s_j, M will adapt itself by adding 1 to each m_i for each i, as described by Equation 10.

$$j \le i < n \qquad \text{[Eq 10]}$$

The third step in creating f is to define a working range r initially having a low end and a high end as shown in Equations 11, 12, and 13.

$$r_{lo} = 0.0 \qquad \text{[Eq 11]}$$

$$r_{hi} = 0.1111111111\ldots \qquad \text{[Eq 12]}$$

$$r = r_{hi} - r_{lo} \qquad \text{[Eq 13]}$$

Fourth, we must define storage for f. As explained above, f cannot be expressed in decimal notation, and instead must be expressed in binary notation, so that rules out floating point storage for f. Integer storage can accommodate only a limited number of bits, so that too must be ruled out. AACTOR uses a string variable for f because strings can be very long and it is easy to append to them. String f_{str} is initialized to empty. When the compressor identifies a bit to be included in f, the bit is converted to a string of length one containing either "0" or "1."The length-one string is appended to f_{str}. When the compressor finishes creating f_{str}, RTTY-AAC processes it left to right it by modulating a Space audio tone for each length-one string "0" and a Mark audio tone for each length-one string "1" (Marks and Spaces are explained later).

Fifth, enter the compression loop, where the compressor iterates through the symbols in the message, starting with the left-most symbol. Note that an [EOT] symbol must be appended to every message to be compressed, or the decompressor will not know when to stop decompressing. Pseudo code for the compression loop is shown in Figure 3.

As explained earlier, these calculations are performed with binary notation, which requires integer variables. After just a few symbols are processed, however, the precision needed for processing the remaining symbols in the message will exceed the precision limits of integer variables. For that reason, the binary notation calculations are performed with a clever integer technique using 32-bit unsigned integer variables as explained in Notes 2, 3, and 4.

As the symbols in the message are processed by the compressor, r_{lo} and r_{hi} will converge. Their most-significant (left-most) bits (MSBs) will often come to be matched and, if so, will not change thereafter and will no longer contribute to the precision of the calculations. The string version of the matching MSBs is appended to f_{str}. The compressor then processes any pending underflow bits (discussed next), left shifts r_{lo} and r_{hi} one bit, and sets the least-significant (right-most) bit (LSB) of r_{hi} to one. This processing of matching most significant bits will occur multiple times during compression.

As symbols in the message are processed by the compressor, and as r_{lo} and r_{hi} converge, they may be converging from below and above towards ½ (0.1 in binary notation), in which case their MSBs will never match and the compression will break down. When this "underflow condition" threatens, the compressor increments a running count of the underflow conditions, changes the next-to-MSBs in r_{lo} and r_{hi} to zero and one respectively, left shifts both r_{lo} and r_{hi} one bit, and sets the least significant bit of r_{hi} to one. After the next occurrence of matching MSBs in r_{lo} and r_{hi}, and after the string version of that matching MSB has been appended to f_{str}, the underflow bit(s) are processed. Based on the running count, each underflow bit is assigned a value opposite to the value of the matching MSBs. Their string versions are appended to f_{str}.

It is the accumulation of matching MSBs and underflow bits that comes to comprise f. In a broad sense, bits flow into r_{lo} and r_{hi} from the right, flow through r_{lo} and r_{hi} from right to left, are modified along the way by the arithmetic, and then are shifted out on the left. Thus, at any given time, r_{lo} and r_{hi} hold only a portion of what will turn out to be f. This is how a theoretically infinitely long f can be processed with fixed-length integer variables.

Finally, after all symbols in the message have been processed, there are a few bits remaining in r_{lo} that are needed to complete f, so they are "flushed" out and their string versions are appended to f_{str}.

AAC Decompression

AACTOR's decompressor initializes f_{str} to empty. As each Space and Mark tone is demodulated by RTTY-AAC, a length-one string "0" or "1," respectively, is appended to f_{str}. (Marks and Spaces are explained later). When the incoming signal goes quiet, it is assumed f_{str} is completed and the AAC decompressor then processes f_{str} left to right. Alternatively, the decompressor could be programmed to process one bit at a time as each bit is demodulated by RTTY-AAC. In either case, as soon as the decompressor identifies an [EOT] symbol decompression ceases.

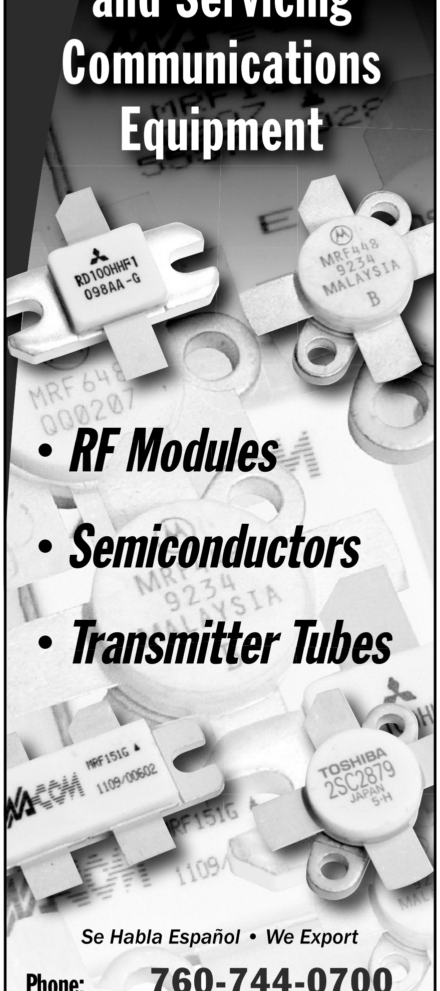
As explained above, an AAC decompressor must use the same S, with the s_i in the same order, as was used by the AAC compressor. The decompressor must also use the same M, with the m_i in the same order. When it initializes, the decompressor knows nothing about the message, other

From MILLIWATTS To KILOWATTS℠

More Watts per Dollar℠

In Stock Now!

Semiconductors for Manufacturing and Servicing Communications Equipment



- ***RF Modules***
- ***Semiconductors***
- ***Transmitter Tubes***

Se Habla Español • We Export

Phone:	**760-744-0700**
Toll-Free:	**800-737-2787**
(Orders only)	**800-RF PARTS**
Website:	**www.rfparts.com**
Fax:	**760-744-1943**
	888-744-1943
Email:	**rfp@rfparts.com**

VISA MasterCard AMERICAN EXPRESS Cards

RF PARTS™ COMPANY

From Milliwatts to Kilowatts℠

```
$r_{lo} = 0.0$
$r_{hi} = 0.1111111111\ldots$
$w$ = bit versions of first 32 length-one strings in $f_{str}$
do
        $r = r_{hi} - r_{lo} + 1$
        $i = w - r_{lo} + 1$
        $i = (i \cdot m_{n-1}) - 1$
        $i = i \div r$
        get $s_i$ from S
        if $s_i$ equals [EOT]
            exit outer loop
        display $s_i$
        $r_{hi} = r_{lo} + (r \cdot hi_i) - 1$
        $r_{lo} = r_{lo} + (r \cdot lo_i)$
        do
            if $r_{hi}$ and $r_{lo}$ MSBs match
                process the MSBs (explained in text)
                process any pending underflow bits (explained in text)
                update w (explained in text)
            else if an underflow condition threatens in $r_{hi}$ and $r_{lo}$
                remove underflow bits (explained in text)
                update w (explained in text)
            else
                update (adapt) M based on i
                exit inner loop
            end if
        loop
loop
```

Figure 4 —This is the Pseudo code for the AACTOR decompression loop.

	RTTY Baudot	AACTOR	
Message	*Bit Count*	***f*** *Bit Count*	*Resulting Size*
QSL?[EOT]	48	28	58.33%
NR?[EOT]	40	23	57.50%
CQ TEST K0JJR CQ[EOT]	176	91	51.70%
W1AW 599 JOE MN W1AW[EOT]	240	109	45.42%
W1AW TU K0JJR CQ[EOT]	192	92	47.92%
RYRYRYRYRY TESTING DE K0JJR[EOT]	264	141	53.41%
A[EOT]	16	12	75.00%
AAAAAAAAAA[EOT]	88	40	45.45%
AB[EOT]	24	17	70.83%
ABABABABABABABABABAB[EOT]	168	76	45.24%
A 0 B 1 C 2 D 3 E 4 F 5 G 6 H 7 I 8 J 9 [EOT]	568	183	32.22%
NOW IS THE TIME FOR ALL GOOD MEN TO COME TO THE AID OF THEIR COUNTRY[EOT]	672	328	48.81%
THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG[EOT]	416	237	56.97%
Lincoln's Gettysburg address (excluding punctuation)	13664	5848	42.80%
strings generated with random symbols from ***S***			
10 symbols	128	62	48.44%
100 symbols	1152	565	49.05%
250 symbols	2984	1377	46.15%
500 symbols	5824	2706	46.46%
1000 symbols	11496	5379	46.79%

Figure 5 — Here are some sample messages, along with the RTTY Baudot bit count and the AACTOR bit count for each message. You can see that the AACTOR bit counts are always significantly fewer than the RTTY Baudot counts.

than that at least one s_i will appear at least once in the message. Because the decompressor does not know which s_i it will be, the decompressor initially must assume that all s_i will appear in the message at least once, even if that assumption later turns out to be incorrect as to some or even most of the s_i. Thus, when the decompressor initializes, the m_i will be the same as when the compressor initialized (Equations 5 and 9, and Figure 2). Similar to what happens during compression of a message, during decompression, M models the message being decompressed by adapting itself as it identifies each symbol in the message.

For the decompressor, r, r_{hi}, r_{lo}, hi_i, and lo_i are defined and initialized in the same way they are defined and initialized for the compressor (Equations 6, 7, 8, 11, 12, and 13). The decompressor also requires a 32-bit unsigned integer working variable, w. The working integer variable w is loaded with the bit versions of the first 32 length-one strings in f_{str}. If less than 32 Mark and Space tones were demodulated, resulting in f_{str} containing fewer than 32 length-one strings, w is right-padded with zero bits. Pseudo code for the decompression loop is shown in Figure 4.

As was true during compression, during decompression the decompressor watches for matching most significant bits in r_{lo} and r_{hi}. When it occurs, the MSBs are left-shifted out of r_{lo} and r_{hi} and discarded, and the least significant bit (LSB) of r_{hi} is set to one. Also, w is left-shifted one bit and its LSB is set to the bit version of the next length-one string in f_{str}. This processing of matching MSBs will occur multiple times during decompression.

Underflow arithmetic will be required multiple times during decompression, as it was during compression. The decompressor addresses a threatened underflow condition in r_{lo} and r_{hi} the same way as the compressor, except that the decompressor discards the underflow bits removed from r_{lo} and r_{hi} and does not keep track of them. Also, the next-to-MSB bit in w is toggled, w is then left-shifted one bit, and the w LSB is set to the bit version of the next length-one string in f_{str}.

In a broad sense, the bits in the demodulated compressed message flow into w from the right, flow through w from right to left, are modified along the way by the arithmetic, and then are shifted out on the left. Thus, at any given time, w holds only a portion of the bits in the demodulated compressed message. Again, this is how a theoretically infinitely long f can be processed with fixed-length integer variables.

RTTY Basics

RTTY is based on the five-bit Baudot code named for its inventor, Jean-Maurice-Emile Baudot, a French telegraph engineer.[7] With five bits, up to 32 symbols can be represented ($2^5 = 32$). If two of the Baudot code symbols are dedicated to "shift" and "unshift" symbols, up to 60 symbols can be represented [$(2^5 - 2) \times 2 = 60$]. When the demodulating station encounters a shift symbol, all of the Baudot codes following it are treated as shifted Baudot codes, which correspond to the symbols 0 through 9, plus a variety of other special symbols. When the demodulating station encounters an unshift symbol, all of the Baudot codes following it are treated as unshifted Baudot codes, which correspond to the symbols A through Z (upper case only). In RTTY parlance, the shift symbol is called "Figures" ([FIGS]) and the unshift symbol is called "Letters" ([LTRS]).

When modulated as an RTTY Baudot code, each five-bit Baudot code is preceded by one start bit, which is always a zero, and is followed by one or more stop bits (usually two), which are always ones.Thus, a total of eight bits comprise each RTTY Baudot code. Each RTTY Baudot code bit is immediately modulated on the heels of the preceding RTTY Baudot code bit. After the eighth bit in a RTTY Baudot code is modulated, the first bit of the next RTTY Baudot code is immediately modulated. Therefore, an RTTY message is simply a fixed-length stream of binary ones and zeroes. The embedded start and stop bits synchronize the RTTY demodulating station to each RTTY Baudot code.

RTTY employs a binary modulation and demodulation protocol using either frequency shift keying (FSK) or audio frequency shift keying (AFSK). For FSK, the modulating station keys the transmitter at any given frequency for an RTTY Baudot code *one* bit. The modulating station keys the transmitter 170 Hz below that frequency for an RTTY Baudot code *zero* bit. In lower sideband mode, the demodulating station will hear audio tones at 2125 Hz and 2295 Hz below the VFO setting. Lower sideband, the 170 Hz separation, and the 2125 Hz and 2295 Hz audio frequencies are Amateur Radio specifications by agreement, although other specifications are possible and are sometimes used.

For AFSK, which also uses lower sideband, an RTTY Baudot code *one* bit is modulated as an audio tone at 2125 Hz below the operating frequency. An RTTY Baudot code *zero* bit is modulated as an audio tone at 2295 Hz below the operating frequency. At the demodulating station, it cannot be determined whether the modulating station is using FSK or AFSK, and it makes no difference. The demodulating station will demodulate in the same way in either case.

The audio tones at 2125 Hz and 2295 Hz are called "Mark" and "Space" audio tones

From MILLIWATTS *To* KILOWATTS

More Watts per Dollar

Transmitting & Audio Tubes

COMMUNICATIONS
BROADCAST
INDUSTRY
AMATEUR

Immediate Shipment from Stock

3CPX800A7	4CX1000A	810
3CPX1500A7	4CX1500B	811A
3CX400A7	4CX3500A	812A
3CX800A7	4CX5000A	833A
3CX1200A7	4CX7500A	833C
3CX1200D7	4CX10000A	845
3CX1200Z7	4CX15000A	6146B
3CX1500A7	4CX20000B	3-500ZG
3CX3000A7	4CX20000C	3-1000Z
3CX6000A7	4CX20000D	4-400A
3CX10000A7	4X150A	4-1000A
3CX15000A7	572B	4PR400A
3CX20000A7	805	4PR1000A
4CX250B	807	*...and more!*

Se Habla Español • We Export

Phone:	760-744-0700
Toll-Free: (Orders only)	800-737-2787 RF PARTS
Website:	www.rfparts.com
Fax:	760-744-1943 888-744-1943
Email:	rfp@rfparts.com

RF PARTS™ COMPANY

in RTTY parlance. For both FSK and AFSK, the Mark and Space tones must have the same duration. With FSK, the modulating station must key the transmitter for that duration. With AFSK, the modulating station must sound the audio tone for that duration. Duration depends on the baud. Although RTTY recognizes several bauds, the most popular is 45.45, with some activity also at 75. At 45.45 baud, the duration of each Mark and Space audio tone is given by Equation 14.

$$1 \div 45.45 = 0.022 = 22 \text{ milliseconds} \quad \text{[Eq 14]}$$

The rapid mixture of Mark and Space tones gives RTTY its distinctive warbling sound.

RTTY-AAC

Because an RTTY message is simply a fixed-length stream of binary ones and zeroes, and because an AAC-compressed message is likewise nothing more than a fixed-length stream of binary ones and zeroes, RTTY is well-suited for processing AAC-compressed messages. The ones and zeroes in an AAC-compressed message are modulated and demodulated as Mark and Space tones, respectively, just as is the case with the ones and zeroes in an RTTY message. But, RTTY requires some modification to process AAC-compressed messages. Start and stop bits are indispensable to RTTY, and RTTY performs table lookups of Baudot codes, all of which are irrelevant to an AAC-compressed message. RTTY would have to be modified so that it does nothing more than modulate and demodulate a fixed-length stream of binary ones and zeroes. These modifications distinguish RTTY-AAC from RTTY.

On the modulation end, the AAC compressor creates a fixed-length stream of binary ones and zeroes from the original message and turns the stream over to RTTY-AAC for modulation. RTTY-AAC uses any baud supported by RTTY, RTTY Mark and Space tone frequencies, and any Mark and Space duration supported by RTTY (for example, 22 milliseconds at 45.45 baud). Also, like RTTY, RTTY-AAC can be modulated using either FSK or AFSK. On the demodulating end, RTTY-AAC demodulates the stream and turns it over to the AAC decompressor for recreation of the original message.

Speed Comparisons

To determine the speed of AACTOR as compared to the speed of RTTY, the number of bits generated by each mode for the same message must be compared. For AACTOR, the number bits in ***f*** is determined simply by counting them. For RTTY, however, it is not so obvious. At first glance, the RTTY Baudot bit count appears to be simply the number of symbols in the message multiplied by eight bits per symbol. But the [FIGS], [LTRS], and [SP] symbols change the way the RTTY Baudot bits are counted. Consider this message:

W1AW TU UR 599 MN K0JJR

This message seems to have 23 symbols (don't forget to count the spaces), which would be 184 RTTY Baudot bits (23 × 8). To RTTY, however, the message actually looks like this:

[LTRS]W[FIGS]1[LTRS]AW[LTRS]
[SP]TU[LTRS][SP]UR[LTRS][SP]
[FIGS]599[LTRS][SP]MN[LTRS][SP]
K[FIGS]0[LTRS]JJR

In addition to the [FIGS] and [LTRS] symbols, notice that every [SP] symbol is accompanied by a [LTRS] symbol due to the RTTY "unshift on space" protocol. Thus, to RTTY, the message has not 23 symbols, but rather 34, and not 184 RTTY Baudot bits, but rather 272 (34 × 8).

With the bits counted this way, many speed comparisons were simulated on a computer using common RTTY messages, prose messages, short and long messages, messages with few and many different symbols, and messages composed of random symbols. In all cases, the bit count for AACTOR was significantly less than the RTTY Baudot bit count for the same message. Figure 5 shows some examples ([LF] and [CR] are ignored).

As general propositions, the more often that RTTY switches back and forth between [FIGS] and [LTRS], the more often the [SP] symbol appears, and the longer the RTTY message as measured by bit count, the more favorably AACTOR compares to RTTY. As another general proposition, for typical RTTY contest messages, the AACTOR compressed message has approximately one-half the number of bits as compared to the same RTTY message. An RTTY contest message 200 bits long (approximately 17 to 20 symbols) and modulated at 45.45 baud will have a duration of 4.4 seconds (200 × 0.022 seconds). The same message will have a duration of only approximately 2.2 seconds with AACTOR. At 75 baud, the message will have a duration of only approximately 1.3 seconds with AACTOR (200 ÷ 75 ÷ 2).

Practical Considerations

RTTY is self-synchronizing. An RTTY demodulator is constantly on the lookout for start and stop bits to determine when the five-bit Baudot codes start and stop. If bits are lost due to fade or interference, the RTTY demodulator can re-synchronize within the next succeeding few symbols. This self-synchronization is why a demodulating station can tune to an RTTY message in progress and process the message from that point forward. If, however, the lost symbol was a [FIGS] or [LTRS], the succeeding symbols will still be valid symbols, but their message will be nonsensical. For example, a lost [FIGS] symbol preceding the symbols 123456 will cause those symbols to appear on the demodulating end as the unshifted symbols QWERTY. When the next [FIGS] or [LTRS] is encountered, the demodulating station will resynchronize and continue processing the message correctly.

AACTOR cannot self-synchronize like RTTY. If bits are lost due to fading or interference, the balance of the message likely will be gibberish. There will be no opportunity to resynchronize. RTTY is a robust mode, however, not easily susceptible to fading or interference, and RTTY-AAC, when it goes out over the airways, is identical to RTTY and will benefit from that robustness. Also, an AAC-compressed message has significantly fewer bits than an RTTY message, thus reducing the opportunities for fade or interference. Risk of losing bits occasionally is more than outweighed by the improved speed of every message.

RTTY is better suited for real-time keyboard-to-keyboard rag chewing than AACTOR. RTTY can modulate and demodulate symbol-by-symbol with each key press at the modulating station. While the modulating station pauses in its key presses, RTTY will automatically send repetitive [LTRS] symbols (called "diddles" in this context) to keep the two stations synchronized. AACTOR, on the other hand, modulates and demodulates entire messages at a time without interrupting pauses. Rag chewing is possible, but the demodulating station must wait for the modulating station to finish all key presses and to send the entire message. For contesting with AACTOR, this is not an issue. Contesters compose their messages ahead of time and, using macros, send them with one key press or one mouse click. During contesting, there is very little real-time keyboard-to-keyboard messaging.

Conclusion

AACTOR, the combination of AAC and RTTY-AAC, results in a faster version of RTTY that exploits the robustnessof RTTY. By compressing the message to be modulated, fewer Mark and Space tones need to be modulated and demodulated to exchange the message, meaning that it will take less time to exchange messages.

Joseph J. Roby, Jr, is an Amateur Extra Class licensee, KØJJR. He has been active in Amateur Radio since 2002, with emphasis on contesting, DXing, and Amateur Radio soft-

ware development, including writing his own logging and RTTY contesting programs. His software work has been twice published previously, in the February 2005 edition of Popular Communications *("Homebrewing Software for a Computer-Controlled Radio") and in the September 2014 edition of* The 33rd ARRL and TAPR Digital Communications Conference Proceedings *("A Radioteletype Over-Sampling Software Decoder for Amateur Radio"). He is a member of the ARRL, the Minnesota Wireless Association, and the Arrowhead Radio Amateurs Club.*

Joe is a practicing lawyer located in Duluth, Minnesota. His practice focuses on labor and employment law and media law. He has a Bachelor of Science degree in mathematics, with highest honors, from the South Dakota School of Mines & Technology in Rapid City, South Dakota and a Juris Doctorate degree from William Mitchell College of Law in St. Paul, Minnesota. He participates in the ARRL's Volunteer Counsel Program.

Notes

[1]Various contributors, "Radioteletype," 9 November 2014: **http://en.wikipedia.org/wiki/Radioteletype**.

[2]Michael Dipperstein, Arithmetic Code Discussion and Implementation, November 2014:**http://michael.dipperstein.com/arithmetic/**.

[3]Mark Nelson,"Data Compression with Arithmetic Coding," 19October2014:**http://marknelson.us/2014/10/19/data-compression-with-arithmetic-coding/**.

[4]Mark Nelson and Jean-Loup Gailly, "Huffman One Better: Arithmetic Coding," in *The Data Compression Book*, 2nd edition, New York, M&T Books, 1996, Chapter 5, pp 113 – 152. This book is available as a PDF file at **http://lib.mdp.ac.id/ebook/Karya%20Umum/The-Data-Compression-Book.pdf**.

[5]Various contributors,"Arithmetic Coding," 19 November 2014: **http://en.wikipedia.org/wiki/Arithmetic_coding#Adaptive_arithmetic_coding**.

[6]Static (non-adaptive) arithmetic coding works differently. A pass is made through the entire message to be compressed to construct the full and final version of ***M*** before the message is compressed. Using this ***M***, the message is then compressed, and ***M*** remains unchanged (static) during compression. Then, after the message is compressed, ***M*** must be sent as a preface to the compressed message because the decompressor will need this ***M*** to decompress the message. For very long messages, ***M*** is relatively short as compared to the length of the compressed message itself. For short RTTY-like messages, ***M*** is relatively large as compared to the length of the compressed message itself, thus defeating the purpose of creating a faster version of RTTY. The adaptive feature of AAC dispenses with any need to send anything other than the compressed message itself.

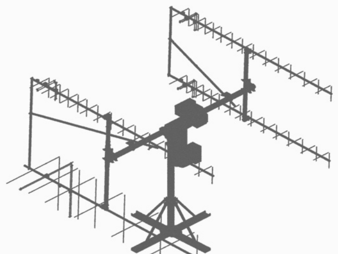
[7]Various contributors,"Baudot Code," 6December2014: **http://en.wikipedia.org/wiki/Baudot_code**.

M² ANTENNA SYSTEMS, INC.

WORLD CLASS PRODUCTS

Bring Your Ideas to Us

M² brings your antenna designs to life!



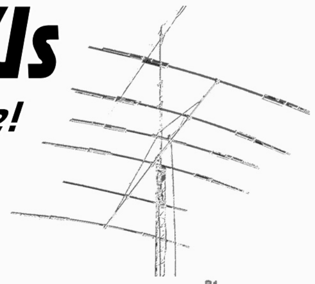
M² makes more than just high quality off-the-shelf products. We also build custom antenna systems using innovative designs to meet our customers' demanding specifications.

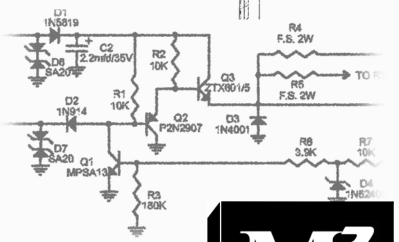
Our high-performing products cover high frequency, VHF, UHF and microwave.
Ask us about our custom dish feeds.

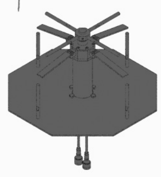
From simple amateur radio installations to complete government and commercial projects, we have solutions for nearly every budget.

Directional HF and small satellite tracking stations are our specialties.

Contact us today to find out how we can build a complete antenna system to meet your needs!







M² ANTENNA SYSTEMS, INC.

M² offers a complete line of top quality amateur, commercial and military grade antennas, positioners and accessories. We produce the finest off-the-shelf and custom radio frequency products available.

For high frequency, VHF, UHF and microwave, we are your source for high performance RF needs. M² also offers a diverse range of heavy duty, high accuracy antenna positioning systems.

For communications across town, around the world or beyond, M² has World Class Products *and Engineering Services* to suit your application.

M² products are proudly 'Made in the USA'

4402 N. Selland Ave.
Fresno, CA 93722
Phone (559) 432-8873
http://www.m2inc.com
sales@m2inc.com

ANTENNAS POSITIONERS ACCESSORIES

Maynard A. Wright, W6PAP

6930 Enright Dr, Citrus Heights, CA 95621: **w6pap@arrl.net**

Octave for Angles

W6PAP gives us another Octave *lesson, this time describing some of the unit conversion challenges we have to deal with.*

The engineering community has been plagued by unit conversion problems for many years. In 1492, Columbus miscalculated his position because of, in part, a confusion between Roman and nautical miles.[1] In 1999, the Mars Orbiter failed because of a confusion between metric and English units.[2] The same sort of confusion caused an error in dimensioning a roller coaster axle, which resulted in a crash at Tokyo Disneyland's Space Mountain in 2004.[3]

We Amateur Radio operators may be subject to some of the same sorts of errors while we design circuits or calculate component values. Things may be made worse for us by the nature of some of the mathematical tools we use. Differences between the default dimensions for keystrokes on scientific calculators and for software functions are listed in Table 1. If we are more familiar with one of these tool types than the other, we may be tempted to unconsciously substitute the wrong units when using the other tool, especially when we're thinking about other aspects of a project as we make calculations.

There is considerable potential for confusion between the meanings of LOG in software and on calculator keys, but the magnitude of the natural logarithm is more than two times the magnitude of the common logarithm for any argument. The difference should be noticeable and should alert us that something is wrong. Still, significant errors have gone unnoticed by skilled engineering teams, causing costly failures of space- and land-based vehicles and systems (see Notes 2 and 3).

More subtle errors are possible when working with trigonometric functions. We'll use the sine for our example here, but the same concerns apply to other trig functions.

Instead of maintaining a large constant error, as in the case of logarithms, the substitution of a value in radians instead of degrees as the argument to a trig function will cause an error whose value cycles with the magnitude of the intended quantity. *Octave* code to plot the correct curve and the incorrect curve is listed in Table 2, and the resulting plot is shown in Figure 1.

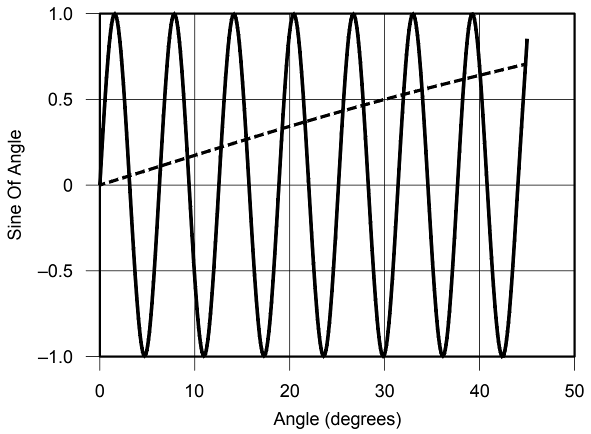
The large number of points specified in

Table 1
Comparison of Calculator and Software Trig Function Conventions

	Function Name For Natural Logarithms	*Function Name For Common Logarithms*
Calculator	ln	log
Software	log	log10 or logten
	Default Argument for Trig Functions	*Default Return for Trig Functions*
Calculator	Degrees	Degrees
Software	Radians	Radians

Table 2
GNU *Octave* Code for Producing Curves in Figure 1

```
x = linspace(0, 45, 300000);
y1 = sin(x);
y2 = sind(x);
# y2 = sin(x * pi / 180);
plot(x, y1, x, y2);
axis([-5, 50, -1.1, 1.1]);
xlabel("Angle (degrees)");
ylabel("Sine Of Angle");
grid();
pause();
```



QX1601-Wright01

Figure 1 — This graph is a comparison of incorrect sin(x) with correct sind(x) when the abscissa is intended to be in degrees.

[1]Notes appear on page 21

the linspace() call is intended to provide a relatively smooth plot at the tops and bottoms of the sinusoids representing the errored curve and isn't really essential to the purpose of the plots. Rather than calculate the maximum number of elements that our display can handle, we've simplified things by using a number that represents overkill.

We've used the function sind() in Figure 1 to calculate values of y2 (in degrees). The commented out line shows a form that may be used when only functions designed to accept arguments in radians are available.

In Figure 1, let's assume that we want to calculate the sine of an angle in degrees somewhere between 0° and 45°. The gradually increasing line represents a plot of the sines of angles in that range on the ordinate, as the function of angles in degrees on the abscissa. The multiple cycle sinusoid represents what happens when we input the angle in degrees to the sin() function in GNU *Octave*, *Matlab*, *Python*, *C*, *C++*, or any one of many other software tools.[4, 5, 6] The software is expecting the argument in radians rather than degrees and returns a value accordingly.

Unlike the case with the logarithm, there are multiple ranges of angles where the correct value of the sine and the errored value are close enough to each other to elude intuitive detection, but are far enough apart to cause a flaw in a precise calculation. This would seem to provide a significant potential for serious calculation errors that cannot be detected by good engineering judgment or common sense.

Many calculators provide a keystroke option for changing to angular input and/or output to radians. It's a little more difficult in software, though, to make the opposite change. Some math utilities such as GNU *Octave* (see Note 4), though, have provided functions to do just that. In recent revisions of *Octave*, we can append a lower case "d" to a trig function and it will expect inputs or produce outputs in degrees. *Python* (see Note 6) does not provide functions for use with degrees, although it provides functions for converting radians to degrees and vice versa.

Let's consider, for example, Equations 83 and 84 on page 2.45 and 2.46 of the 2016 *ARRL Handbook*.[7]

$$X = |Z| \times \sin\theta \,(\text{ohms}) \qquad \text{[Eq 83]}$$

$$R = |Z| \times \cos\theta \,(\text{ohms}) \qquad \text{[Eq 84]}$$

The *Handbook* illustration converts an impedance of $Z = |12.0\ \Omega|\ \underline{\angle -42^\circ}$ into its rectangular components, *R* and *X*.[8] We'll plug the angle –42° into the function sin() and cos() in *Octave*, and we'll see a significant error that will mentally flag us that

Table 3
Rectangular Components of $|12\ \Omega|\ \underline{\angle -42^\circ}$

Correct Values	*Incorrect Values*	*Error*
$R = 9.82\ \Omega$	$R = -4.80\ \Omega$	154%
$X = -8.03\ \Omega$	$X = 11.0\ \Omega$	237%

Table 4
Rectangular Components of $|12\ \Omega\,|\ \underline{\angle -32^\circ}$

Correct Values	*Incorrect Values*	*Error*
$R = 10.2\ \Omega$	$R = 10.0\ \Omega$	*1.96%*
$X = -6.36\ \Omega$	$X = -6.62\ \Omega$	*4.09%*

something is wrong. In addition to the difference in magnitudes of the sine and cosine, the signs of both returns will be incorrect. The correct and incorrect rectangular components are listed in Table 3.

The enormous percentage error reflects the change in magnitude, but even more so the change in sign, and indicates that any downstream calculation in which we use these numbers will be badly flawed. If, though, we are paying any attention to the nature of the calculations, the error ought to be apparent.

We can see the error in Figure 1 by observing both curves for an abscissa of 42°.[9] Note that the ordinates of the two curves at that point are on opposite sides of the zero line.

Let's leave the magnitude at 12.0 Ω but change the angle to –32° to see what happens. Here the values are much closer to each other, as shown in Table 4.

These are much smaller, but still significant, errors, especially for *X*. If we are working on a problem that requires a precision of three or more significant figures, these errors are certainly too high. They are not so high, though, as to raise an alarm in the minds of most of us. The solution here is to be very cautious, especially when moving back and forth between a calculator and software.

Returning to Figure 1, we can see that, in agreement with the calculations above, the two curves are much closer to each other for an abscissa of 32°. Although we haven't plotted the cosine function as we did the sine function, we would find a similar situation if we did.

The same concerns apply to all the trig functions, including the tangent and cotangent functions. As those two functions feature periodic vertical asymptotes with infinite discontinuities, though, plots of those functions would not be as helpful in understanding the problem as is Figure 1.

Maynard Wright, W6PAP, was first licensed in 1957 as WN6PAP. He holds an FCC General Radiotelephone Operator's License with Ship Radar Endorsement, is a Registered Professional Electrical Engineer in California, and is a Life Senior Member of IEEE. Maynard was involved in the telecommunications industry for over 48 years. He has served as technical editor of several telecommunications standards and holds several patents. He is a Past Chairman of the Sacramento Section of IEEE. Maynard is an ARRL Member. He is Secretary Treasurer and Past President of the North Hills Radio Club in Sacramento, California.

Notes

[1]For 6 stories about miscalculations, go to: **http://mentalfloss.com/article/25845/quick-6-six-unit-conversion-disasters.** Columbus' error is a navigational error rather than an engineering error, but serves to indicate that such unit errors are not confined to our era.

[2]Kathy Sawyer, Staff Writer, "Mystery of Orbiter Crash Solved," Washington Post, October 1, 1999: **www.washingtonpost.com/wp-srv/national/longterm/space/stories/orbiter100199.htm**.

[3]For more unit conversion errors, including the story about the Tokyo Disneyland roller coaster crash, see "Unit mixups — Colorado State University," **lamar.colostate.edu/~hillger/unit-mixups.html**, and **spacemath.gsfc.nasa.gov/weekly/6Page53.pdf**.

[4]You can learn more about GNU *Octave*, and download the latest version of *Octave* at: **www.octave.org**.

[5]For information about *Matlab*, and to download a trial version of the software, go to: **www.mathworks.com/products/matlab**.

[6]Learn more about Python and download the software at: **www.python.org**.

[7]H. Ward Silver, NØAX, Ed. *The ARRL Handbook for Radio Communications*, 2016 Edition, ARRL, 2015, Equations 83 and 84 on pp 2.45 – 2.46. ISBN: 978-1-62595-041-3; ARRL Publication Order No. 0413, $49.95. ARRL publications are available from your local ARRL dealer or from the ARRL Bookstore. Telephone toll free in the US: 888-277-5289, or call 860-594-0355, fax 860-594-0303; **www.arrl.org/shop**; **pubsales@arrl.org**.

[8]We'll round to three significant figures for the purposes of this article.

[9]Although Figure 1 plots the sine in degrees in the first quadrant, for clarity, and the ARRL Handbook example involves angles in the fourth quadrant, the identity sin(–x) = –sin(x) allows the use of Figure 1 to compare the returns of sin() and sind() for any angle in the fourth quadrant with a reversal of the signs of the returns. You can do this mentally by prefixing a negative sign to every Y axis value in Figure 1 while considering the example. If we were to plot cos() and cosd() curves, the identity cos(x) = cos(–x) would indicate that no sign reversal is required when moving between the first and fourth quadrants.

DX ENGINEERING

8:30 am to midnight ET, Monday-Friday
1230 to 0400 UTC March-October

8:30 am to 5 pm ET, Weekends
1230 to 2100 UTC March-October

International/Tech: 330-572-3200
8:30 am to 7 pm ET, Monday-Friday
Country Code: +1 Sale Code: 1601QEX

ORDER BY 10 PM SAME-DAY SHIPPING
Mon-Fri, 10 pm ET
In-Stock Items

800-777-0703 | DXEngineering.com

Sign up at DXEngineering.com. Get news & specials by email. Request a FREE Catalog.

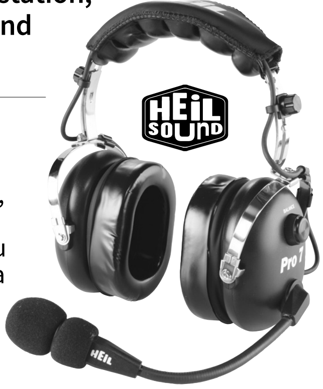
From the microphone to the tower, DX Engineering has the gear to build a competitive station, along with the best support and fastest shipping in the industry. That means you'll spend more time on the air and less time waiting on your gear to arrive.

Receive Preamplifiers
An advanced push-pull design eliminates harmonic distortion to make these the best low-noise preamps on the market—even better than many radio front-ends. With 300 kHz-35 MHz coverage, they are perfect for AM and HF DXing, SWL or contesting. These preamps handle strong signals without overloading.

Antenna Receive Interface
Connect an external receive-only antenna to your transceiver with this interface. It can also be used as a TX/RX switch for older gear, to insert a receive preamplifier, or to interface to phasing/noise cancelling systems.

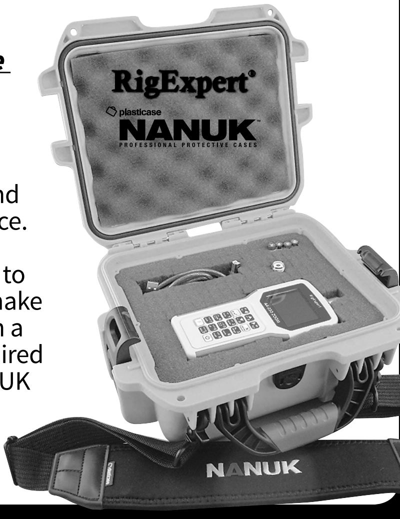
Receiver Guard 5000 and 5000HD
Protect your receiver against potentially dangerous levels of RF from nearby transmit signals. They work automatically and won't affect station performance. Select the original Receiver Guard, or the new HD version made for upper-tier, high-performance radios.

Pro 7 Headsets
With sonically-insulated ear pads and an innovative acoustic design, these headsets block out all the background noise, leaving just you and your radio. They also feature a unique phase reversal switch to really dig out weak signals.



Receive Antenna Variable Phasing Controller
Combine two identical receive antennas to create a "virtual rotator" directional array. It's adjustable, so you can null-out unwanted directional noise or enhance distant signals. This controller is the ideal solution for imperfect antenna setups, giving you great low band reception regardless of your landscape. You're also able to upgrade your NCC-1 with optional filter sets for band-specific enhanced performance.

ZOOM Antenna and Cable Analyzer, and NANUK Equipment Case Combo
The AA-230 ZOOM is easy to use, boasting a large color display, intuitive hotkeys, and straightforward user-interface. The analyzer covers 100 kHz to 230 MHz and can be used to determine return loss and make R/X/Z/L/C measurements on a specific frequency. We've paired it with a perfectly sized NANUK Equipment Case to keep it protected in the field.



Shrink the Globe. *Stay connected:*

Chuck Adams, K7QO

27615 N 130th Ave, Peoria, AZ 85383-2860; **chuck.adams.k7qo@gmail.com**

Crystal Parameter Measurements Simplified

The author describes a procedure to make very accurate measurements on quartz crystals. You can do this with a simple fixture using four resistors, a capacitor and some RF connectors.

This is a technique I derived in 1961 for measuring crystal parameters in a laboratory as an undergraduate student. Fifty years later as radio amateurs, we have much better equipment available on our workbench to do this.

Besides the fixture, the additional equipment needed consists of:

- A digital RF signal generator.
- A frequency counter.
- An RF voltmeter or RF probe.

Crystal Parameters

The quartz crystal unit, in an HC-49U package, consists of a circular quartz disc with aluminum or gold plating on opposite surfaces. The crystal is mounted vertically inside the case. It is held by two supports on the edges of the crystal. Two leads exit the base to secure the crystal in a circuit. Figure 1 is a photo of the internal structure of an HC-49U crystal unit on the left, and the unit in the case on the right.

The quartz crystal unit is electrically represented by a series resistor, R, an inductor, L, and a capacitor, C. A parallel capacitor, C_0, is needed because of the plating and the leads. Figure 2 is the equivalent circuit diagram.

The parameters R, L, and C are referred to in technical publications and books as R_m, L_m, and C_m. The inductance, capacitance and resistance are referred to as the motional parameters of the quartz crystal, thus the subscript *m*.

Derivation of the Resonant Frequency Formulas

The admittance, Y_{AB}, between the terminals A and B in the schematic of Figure 2 is given by Equation 1.

$$Y_{AB} = \frac{1}{Z_{AB}} = \frac{1}{R + j(\omega L - 1/\omega C)} + j\omega C_0 \qquad \text{[Eq 1]}$$

where $\omega = 2\pi f$.

By combining the two terms on the right, we get Equation 2.

$$Y_{AB} = \frac{(1 - \omega^2 LC_0 + C_0/C) + j\omega RC_0}{R + j(\omega L - 1/\omega C)} \qquad \text{[Eq 2]}$$

and inverting both sides gives us Equation 3.

$$Z_{AB} = \frac{R + j(\omega L - 1/\omega C)}{(1 - \omega^2 LC_0 + C_0/C) + j\omega RC_0} \qquad \text{[Eq 3]}$$

Multiplying both the numerator and the denominator by the complex conjugate of the denominator gives us Equation 4.

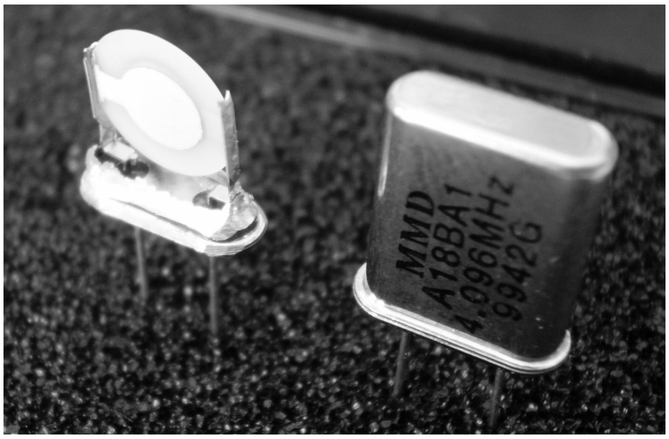


Figure 1 — The internal structure of a quartz crystal unit is shown on the left. The complete package in the metal HC-49U case is shown on the right.

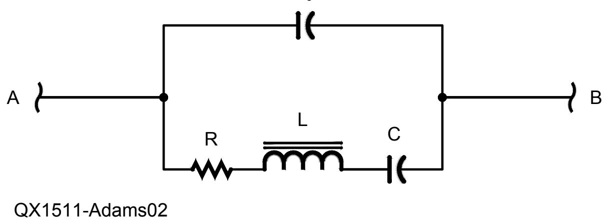


Figure 2 — This is an equivalent circuit for a quartz crystal unit.

$$Z_{AB} = \frac{Real + j(\omega L - 1/\omega C - \omega^3 L^2 C_0 + 2\omega L C_0 / C - C_0 / \omega C^2 - \omega R^2 C_0)}{C_0^2 / C^2 + 1 - 2\omega^2 L C_0 + 2C_0 / C + \omega^2 R^2 C_0^2 - 2\omega L C_0^2 / C + \omega^4 L^2 C_0^2} \quad \text{[Eq 4]}$$

where *Real* is a real number with too many terms to fit on the line. We are not going to use it anyway.

At resonance, the complex component of the above equation is zero. That is the term following the *j*. We can simplify Equation 4 by expressing the complex component as Equation 5.

$$\omega L - 1/\omega C - \omega^3 L^2 C_0 + 2\omega L C_0 / C - C_0 / \omega C^2 - \omega R^2 C_0 = 0 \quad \text{[Eq 5]}$$

Multiplying both sides of the equation by ωC^2 to remove the fractions gives us Equation 6.

$$\omega^2 L C^2 - C - \omega^4 L^2 C^2 C_0 + 2\omega^2 L C C_0 - C_0 - \omega^2 R^2 C^2 C_0 = 0 \quad \text{[Eq 6]}$$

The last term is much smaller than the other terms combined, so we eliminate it. The result is given in Equation 7.

$$\omega^4 L^2 C^2 C_0 - \omega^2 (L C^2 + 2 L C C_0) + (C + C_0) = 0 \quad \text{[Eq 7]}$$

We can solve this equation by finding the roots of the quadratic equation with ω^2 as the independent variable. There are any number of good mathematical software packages that can do this easily. *Wolfram Alpha* is a free online calculator.[1]

There are two resulting resonant frequencies. The series resonant frequency, f_s, is given by Equation 8.

$$f_s = \frac{1}{2\pi}\sqrt{\frac{1}{LC}} \quad \text{[Eq 8]}$$

The parallel resonant, or antiresonant frequency, f_a, is given by Equation 9.

$$f_a = \frac{1}{2\pi}\sqrt{\frac{1}{LC} + \frac{1}{LC_0}} \quad \text{[Eq 9]}$$

We can see that f_a is always greater than f_s.

The crystal is always connected to an external circuit, and C_0 has additional capacitance in parallel with it. We will call that additional capacitance C_p. This will modify Equation 9, and the parallel resonant frequency will be given by Equation 10.

$$f_a = \frac{1}{2\pi}\sqrt{\frac{1}{LC} + \frac{1}{LC_t}} \quad \text{[Eq 10]}$$

where $C_t = C_0 + C_p$. Crystal manufacturers specify the resonant frequency of a crystal at this frequency with a particular load capacitance, C_p.

Impedance at Resonant Frequencies

At the series resonant frequency, f_s, we get $\omega_s^2 = 1/LC$, and by plugging this expression into Equation 3 we get Equation 11.

$$Z_{AB} = \frac{R}{1 - (\omega R C_0)^2} \approx R \quad \text{[Eq 11]}$$

We use the approximation because R is on the order of 10 to 100 Ω, and ω is on the order of 10^6, but C_0 is just a few picofarads, and on the order of 10^{-12}. This makes the term very small compared to 1.

For the the parallel or antiresonant frequency we have Equation 12.

$$\omega_a^2 = \frac{1}{LC} + \frac{1}{LC_t} \quad \text{[Eq 12]}$$

Substituting the ω^2 value into Equation 3, and using the impedance of the capacitor, X_C, we obtain Equation 13.

$$Z_{AB} = \frac{1}{\omega^2 C_t^2 R} = \frac{X_c^2}{R} \quad \text{[Eq 13]}$$

The impedance for the parallel or antiresonant frequency is also pure resistance and much greater than the series resonant impedance, with a value typically between 100 kΩ and 1 MΩ.

In order to obtain C_m and L_m, we need only to measure the series resonant frequency and the antiresonant frequency, and the capacitance, C_0. We then use the numbers in Equations 8 and 10 to solve for L_m and C_m. We need a stable and accurate signal generator, an accurate and precise frequency counter and a fairly sensitive RF voltmeter or RF probe. The frequency counter should be able to measure and display frequencies to within 1 Hz. The frequency counter may be built into the signal generator.

The output level from the test fixture at the parallel resonant point is going to be down as much as 110 dB from the peak voltage. This makes this measurement very difficult. Let's find an easier way.

Crystal in Series With A Capacitor

Let's examine a crystal in series with a capacitor. Figure 3 shows the schematic for this model.

The impedance between terminals A and B of the circuit is given by Equation 14.

$$Z_{AB} = \frac{R + j(\omega L - 1/\omega C)}{1 - \omega^2 L C_0 + C_0 / C + j\omega R C_0} + \frac{1}{j\omega C_X} \quad \text{[Eq 14]}$$

We wade through some lengthy arithmetic to find the two resonant frequencies. This is more tedious than the previous derivation, resulting in an expression with more than 20 terms. I will not bore you with the details and leave it as an exercise, if you are interested in a challenge. The two resulting resonant frequencies are given by Equations 15 and 16.

$$\omega_c = \sqrt{\frac{1}{LC} + \frac{1}{LC_t}} \quad \text{[Eq 15]}$$

$$\omega_a = \sqrt{\frac{1}{LC} + \frac{1}{LC_0}} \quad \text{[Eq 16]}$$

where $C_t = C_0 + C_X$ and $\omega = 2\pi f$.

We have shifted the previous series resonant point, now represented as ω_c, up in frequency. The antiresonant frequency remains exactly the same.

We now use Equations 8 and 15 to determine L_m and C_m of the crystal. This is a system of two equations with three unknowns. We measure C_0 directly with an L/C meter. Take Equation 8 and rewrite it as Equation 17.

$$\omega_s^2 = \frac{1}{LC} \quad \text{[Eq 17]}$$

[1]Notes appear on page 26

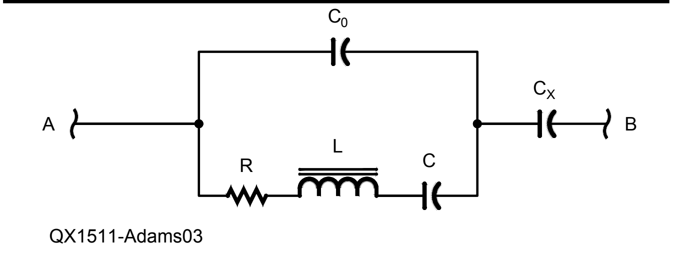


Figure 3 — This schematic diagram is the model circuit for a crystal in series with capacitor C_X.

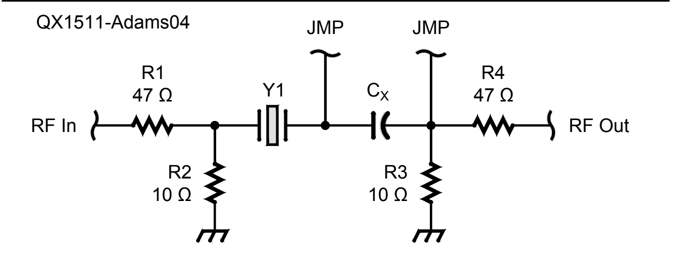


Figure 4 — Here is the schematic diagram for the author's crystal test fixture. R1 = R4 = 47 Ω, R2 = R3 = 10 Ω, Y1 is the crystal under test, and C_X = 47 pF. JMP is a jumper to short out C_X.

We will also rewrite Equation 15 as Equation 18.

$$\omega_c^2 = \frac{1}{LC} + \frac{1}{LC_t} \quad \text{[Eq 18]}$$

The subscript c indicates that this is a measurement with C_X in series with the crystal.

Subtracting Equation 17 from Equation 18 gives us Equation 19.

$$\omega_c^2 - \omega_s^2 = \frac{1}{LC_t} \quad \text{[Eq 19]}$$

This now becomes Equation 20.

$$4\pi^2 (f_c^2 - f_s^2) = \frac{1}{LC_t} \quad \text{[Eq 20]}$$

At this point, everyone wants to make an approximation for the difference of the two squares. Let's use the equation $x^2 - y^2 = (x + y)(x - y)$ and get more precise results. This will give us Equation 21, solved for L_m.

$$L_m = \frac{1}{4\pi^2 (f_c + f_s)(f_c - f_s)(C_0 + C_X)} \quad \text{[Eq 21]}$$

We can measure the two resonant frequencies using a signal generator and frequency counter. Measure C_0 and C_X using a capacitance meter, and then crunch the numbers.

Test Fixture

In order to make the measurements we use a test fixture. Other test measurements in publications and on the Internet use more complex circuits. This test circuit is very simple. Figure 4 shows the schematic diagram for the circuit. You can see how simple and inexpensive it can be.

The input and output impedance of the fixture is close to 50 Ω, but is not critical. R2 and R3 should be kept small to reduce the loaded *Q* on the crystal, but not too small to attenuate the output RF voltage of the fixture to a very small value. The resonant frequencies are not affected by these values. If the values are large, the resonant peak spreads out and it is more difficult to home in on the exact peak. The small values of R2 and R3 also serve to swamp any effects of stray capacitance in the fixture.

Figure 5 is a photograph of the test fixture that I use for this procedure.

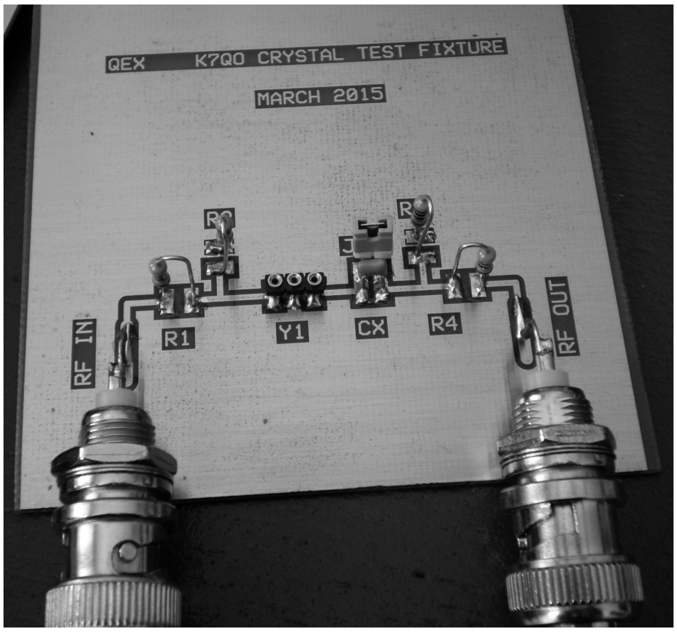


Figure 5 — This photo shows the author's crystal test fixture. It was built using a circuit board layout with parts labeled. The center pin of the crystal socket is grounded to reduce the socket capacitance across the crystal.

Here are the steps to measure the data needed.

1) With the RF generator voltage connected to the fixture, find the series resonant frequency with capacitor C_X shorted. Start the frequency generator a few kilohertz below the marked frequency of the crystal and slowly increase the frequency while watching the RF output level. Write down the frequency at which the peak output voltage occurs. This is f_s.

2) Remove the short across capacitor C_X. Find the new output voltage peak at a slightly higher frequency. This will be f_c.

3) When you build the fixture measure C_X before installing it and again in the circuit without a crystal in the socket to determine the extra stray capacitance caused by the shorting terminals. I used a 47 pF disc capacitor, but any value near this should do nicely. I recommend an NPØ capacitor.

4) Measure C_0 of the crystal using an accurate L/C meter, such as the Almost All Digital Electronics (AADE) L/C Meter II.[2]

Now you have all the data needed to calculate L_m. We obtain C_m from the series resonant frequency, Equation 8, by using the L_m value and f_s.

Crystal Motional Resistance

Here are the steps to measure the motional resistance, R_m, or the effective series resistance (ESR) of the crystal.

1) Short C_X and again find the series resonant frequency. Make note of the output voltage as accurately as possible.

2) Remove the crystal, leaving everything else as is.

3) Replace the crystal with a variable resistor of 100 Ω. I use a 25 turn variable resistor that has 0.2 inch lead spacing, to fit the crystal socket. Adjust the resistor for the exact same RF voltage output we had with the crystal in the socket.

4) Remove the variable resistor and measure the resistance. This is R_m.

Congratulations. You have all four crystal parameters. You have L_m, C_m, R_m, and C_0. These values may be used to determine the circuit for a crystal IF filter with a specific bandwidth.

You can determine the quality factor, Q, of the crystal by taking the series resonant frequency, f_s, motional inductance, L_m, and series resistance, R_s, and use Equation 22.

$$Q = \frac{X_L}{R_m} = \frac{\omega_s L_m}{R_m} = \frac{2\pi f_s L_m}{R_m} \qquad \text{[Eq 22]}$$

This is the inductive reactance at the resonant frequency divided by the motional resistance, R_m, of the crystal.

I have written some *Python* code to perform the calculations for the parameters of the crystal under test. This code carries out the computations to the full 64 bit precision of the computer processor. My code is available for download from the ARRL *QEX* files web page.[3]

Procedure Verification

In order to verify that this procedure is both useful and accurate I picked at random nine crystals from my collection. I then sent these to Tom Thomson, WØIVJ, in Colorado to measure their characteristics by using an AIM Model 4170 Vector Network Analyzer. He also had Larry Benko, WØQE, do the same measurements with another 4170 VNA. Table 1 shows the results, with their measurements and mine. As you can see, the agreement on the crystal parameters is excellent.

SPICE Simulation

As a check of all my theoretical work, I ran a *SPICE* simulation using *ngspice*. I set up an input RF voltage of 1.00 V and swept a crystal model from 4.190 MHz to 4.210 MHz. The voltage output was plotted in dB to show the null depth.

The important thing to note is that the null, corresponding to the parallel resonant mode, remains at the same frequency, but varies in magnitude. This agrees with the theoretical derivation and resulting formula.

Matching Crystals

Using the technique discussed to match crystals is going to be a long and tedious task. One of the things that I want to demonstrate is how a Colpitts crystal oscillator can be used to match crystals and get excellent results.

Suppose you have a number of crystals and you are looking for four crystals for a four pole crystal filter. You want the crystals to match within 10 Hz of each other. Then, using the oscillator and a frequency counter you plug the crystals in and measure the output frequency of each, and keep them ordered. Also note the output voltage from the oscillator. If we have two crystals with the same frequency, we will take the one with the higher output from the oscillator because it will have the lowest R_m.

I have a few hundred 4.096 MHz crystals that I won at an auction on eBay. I found four of them that matched within 5 Hz of each other in the oscillator. I then used the procedure outlined in this paper to measure their crystal parameters. My results are given in Table 2.

Depending upon what program you use to generate the component values for your filters, you can match crystals using the Colpitts crystal oscillator, and then measure the parameters of just one crystal for use in the program. You could also measure a few of the matched crystals and average their parameter values to use in the program. Experimentation will determine which is the fastest method and just how well it meets your criteria for the resulting filter(s).

Conclusion

You now know how to measure crystal parameters accurately and how to easily match a set of crystals for a filter. The test fixture is simple and easy to contruct using any of a number of building techniques. I hope that you will find this test fixture and procedure to be a useful addition to your workbench, and that it will simplify the construction of many successful projects.

Chuck Adams, K7QO, was first licensed as KN5FJZ in the mid 1950s, during the greatest sunspot cycle in recorded history. He has held the calls K5FJZ, K5FO, and now K7QO. He is a retired professor of computer sciences, electrical engineering, and physics. He holds a PhD in physics, with a specialization in radiative transfer and electromagnetics. He now spends his time experimenting and building his own equipment. From time to time, he even gets on the air.

Notes

[1]*Wolfram Alpha* is a free online calculator that will calculate a large variety of quantities from your input values. Go to **www.wolframalpha.com**.

[2]The Almost All Digital Electronics website has had information about an array of kits, including the L/C Meter II at **www.aade.com**. [Unfortunately, when I checked this link prior to publication, the website home page has a note informing us that Neil Heckt passed away on August 19, 2015. The note further indicates that we should be patient while his family determines the future of the company. — *Ed.*]

[3]The author's *Python* code for computing the crystal parameters from the measured data is available for download from the ARRL *QEX* files web page. Go to **www.arrl.org/qexfiles** and look for the file **1x16_Adams.Zip**

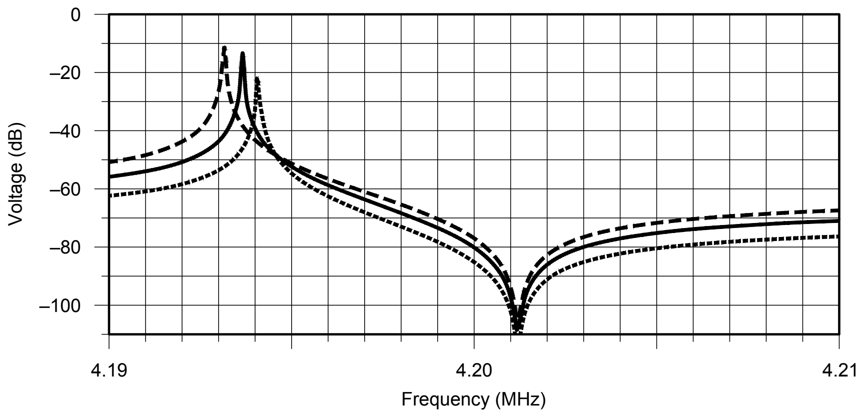


Figure 6 — SPICE simulation for sweeping a crystal. The left-most curve is with no series capacitor and then two more curves for two different values for C_X.

QX1511-Adams06

Table 1
Crystal Measurement Procedure Verification

Lab Tech	*Crystal Number*	*F_{Series}*	*$F_{Parallel}$*	*R_s*	*L_s (mH)*	*C_s (pf)*	*C_p (pf)*	*Q_s*	*Measuring Instrument*
WØIVJ	1	3.578426	3.585154	49.822	139.418	0.0141885	3.780	65801	AIM 4170 VNA
WØQE	1	3.578427	3.585256	49.700	142.449	0.0138866	3.638	64443	AIM 4170 VNA
K7QO	1	3.578426	--------	49.6	141.624	0.013968	3.65	64199	K7QO Fixture
WØIVJ	2	4.193154	4.200966	16.943	111.185	0.0129572	3.484	180122	AIM 4170 VNA
WØQE	2	4.193163	4.200894	17.159	115.719	0.0124495	3.376	177674	AIM 4170 VNA
K7QO	2	4.193159	--------	15.4	114.234	0.012611	3.22	195432	K7QO Fixture
WØIVJ	3	4.031548	4.036547	40.962	309.640	0.0050332	2.032	211124	AIM 4170 VNA
WØQE	3	4.031552	4.036428	40.046	340.051	0.0045830	1.895	215101	AIM 4170 VNA
K7QO	3	4.031553	--------	39.1	326.544	0.004773	1.57	211552	K7QO Fixture
WØIVJ	4	4.193152	4.201202	18.176	107.122	0.0134487	3.509	165524	AIM 4170 VNA
WØQE	4	4.193157	4.201100	18.432	112.633	0.0127907	3.376	160993	AIM 4170 VNA
K7QO	4	4.193154	--------	17.5	112.514	0.012804	3.44	169390	K7QO Fixture
WØIVJ	5	4.094814	4.102963	23.238	134.404	0.0112398	2.830	147508	AIM 4170 VNA
WØQE	5	4.094819	4.103052	23.469	134.440	0.0112368	2.794	147386	AIM 4170 VNA
K7QO	5	4.094819	--------	21.8	130.161	0.0130161	2.74	153616	K7QO Fixture
WØIVJ	6	3.998939	4.005005	22.484	132.716	0.0119351	3.940	153331	AIM 4170 VNA
WØQE	6	3.998953	4.005015	22.619	136.166	0.0116326	3.837	151261	AIM 4170 VNA
K7QO	6	3.998947	--------	21.6	133.566	0.0133566	3.80	155370	K7QO Fixture
WØIVJ	7	11.055203	11.079818	7.407	11.640	0.0178059	4.007	109648	AIM 4170 VNA
WØQE	7	11.055211	11.079788	7.337	11.755	0.0176319	3.965	111282	AIM 4170 VNA
K7QO	7	11.055188	--------	7.1	11.449	0.018102	3.99	112009	K7QO Fixture
WØIVJ	8	4.094873	4.102956	24.034	130.270	0.0115962	2.943	144651	AIM 4170 VNA
WØQE	8	4.094876	4.103023	24.476	134.745	0.0112110	2.818	141641	AIM 4170 VNA
K7QO	8	4.094880	--------	24.1	132.374	0.011412	2.90	161414	K7QO Fixture
WØIVJ	9	13.499968	13.529170	4.063	5.074	0.0273900	6.345	100390	AIM 4170 VNA
WØQE	9	13.499973	13.529200	4.129	5.064	0.0274465	6.339	104041	AIM 4170 VNA
K7QO	9	13.499920	--------	4.10	4.739	0.029329	6.10	98042	K7QO Fixture

Number	*Form Factor*	*Crystal Identification Printed on Each Unit*
1	HC-49U	MPCO 3.579545
2	HC-49U	HOSONIC 4.1943 B603
3	HC-49S	4.032
4	HC-49U	HOSONIC 4.1943 B603
5	HC-49U	MMD A18BA1 4.096JHz 9942G
6	HC-49U	ABRACON 4.000 AB 0443
7	HC-49U	FOX115-20 11.0592
8	HC-49U	MMD A18BA1 4.096MHz
9	HC-49U	78941-1 13.500 KDS 5K

Table 2
Sample Crystal Measurements

Crystal	*f_s (Hz)*	*f_c (Hz)*	*L_m (mH)*	*C_m (fF)*	*C_0 (pF)*
1	4094849	4095292	132.12	11.43	2.97
2	4094849	4095287	133.94	11.28	2.85
3	4094846	4095294	130.82	11.54	2.90
4	4094849	4095301	129.62	11.65	2.92

Scotty Cowling, WA2DFI

PO Box 26843, Tempe, AZ 85285: **scotty@tonks.com**

Hands-On-SDR

In this installment, we move back toward the basics (I didn't say simple!) and delve a bit more into the inner workings of the magical, mystical field programmable gate array, or FPGA. This column relies heavily on what I covered in my Mar/Apr 2015 column.[1] If you have not read that, or can't remember reading it, now would be a good time for a quick review. Since I can't remember writing it, I need to take a short break and read it again myself…

In the Mar/Apr 2015 column, I showed you how to set up an FPGA coding environment with free development tools, walked you through the code of an SDR design example, showed you how to compile the example code and run it on real hardware. We did cover some SDR theory, but we took much of the background as a given and instead focused on how we implemented functions *inside* the FPGA.

This time we will be taking the open-source code written for a variant of the high-performance software-defined radio (HPSDR) Hermes single-board transceiver (specifically the Apache Labs Anan-10e) and port it to the BeMicroCVA9 development board from Arrow Electronics. This board is used in the Hermes Lite project as well as in the IQ2 transceiver and is also compatible with the SDRstick HF1, HF2 and TX2 RF boards. I am going to focus on the HF2 receiver and TX2 transmitter boards, but I will include enough information for you to port the Hermes code to almost any compatible RF front-end board. Given the price and performance of the BeMicroCVA9, I expect that a bevy of hardware designs will surface once the word gets out. So let's start getting the word out!

We owe many thanks to Phil Harman, VK6PH, Kirk Weedman, KD7IRS, and Alex Shovkoplyas, VE3NEA, who wrote the original code that we will use as a starting point. As you look through the code, I think you will be grateful for their work. Without their significant efforts, we would have to write all of this complicated code ourselves!

[1]Notes appear on page 34

What Do We Need to Get Started?

As with each of these columns, limited space begs the questions: "What do I need to know?" and "What equipment do I need?"

You will need a basic working knowledge of the Verilog hardware description language. If you followed my Mar/Apr column, you are prepared enough. We will not be deep diving into the intricacies of the code, since we are just porting the code to a new device. My assumption is that the existing code is working, and we will try not to introduce any new bugs as we port to the new device. Of course, my assumption may prove to be false, but that is a topic for another day: debugging FPGA code!

For hardware, you will need a BeMicroCVA9 development kit.[2] To actually run the code that we are going to compile in this column, you will also need an HF2 board (to receive), or both HF2 and TX2 boards (to transceive).[3, 4] As a lower-performance (and less expensive) alternative, you can use an HF1 or Hermes-Lite board, but you will need to make other modifications to the code if you go that route.[5, 6] I believe that the Hermes-Lite group has ported their firmware to the BeMicroCVA9. After wading through this column, you should be expert enough to compile their source code and run it on the BeMicroCVA9. Even if you do not have the hardware, you can still follow along with the text and learn about porting FPGA code to new devices.

Like my Mar/Apr column, you will need some Verilog programming knowledge, but SDR knowledge in general is not required. We are *targeting* a new device, not *designing code* from scratch.

For design software, there is good news and bad news. The good news is that Altera offers their *Quartus II* FPGA design software as a free download from the Internet for the FPGAs in their Cyclone® family of parts. Both *Linux* and *Windows* versions are available. We will need *two* versions of the *Quartus II* design software to complete the porting work. The first version is *Quartus II* version 13.1, which is the version that was used to create the code that we are going to port. The second version is the latest (as of this writing), version 15.0. The bad news is that *Quartus II* version 15.0 requires a 64-bit operating system. You will need 64-bit *Windows XP*, *Windows 7* or later or 64-bit *Linux* in order to run this new version.

All of the information from my Mar/Apr column applies to both *Quartus* versions. Before you continue, you will need to download and install both of the free *Quartus II* versions (13.1 and 15.0) from the Altera web site.[7] To save some download time, you only need to download Cyclone III and Cyclone V device support for *Quartus II* version 13.1, and only Cyclone V device support for *Quartus II* version 15.0.

Why Two *Quartus* Versions?

Before we get down to the meat and potatoes, I need to explain some problems that we face that are unique to our task. You might ask "Why do we need two versions of *Quartus*?" The answer is tied to the capabilities of each *Quartus* version and the FPGA part that we are migrating *from* as well as the part we are migrating *to*. The Hermes code targets the Cyclone III EP3C25Q240C8 (our *from* part number), while the BeMicroCVA9 uses a Cyclone V 5CEFA9F23C8 (our *to* part number). *Quartus II* version 13.1 supports all Cyclone III parts and some of the Cyclone V parts, but unfortunately not our *to* part number. *Quartus II* version 15.0 supports all Cyclone V parts (including our *to* part number), but no Cyclone III parts at all!

While it is certainly possible to migrate *Quartus* versions and part families in one step (which was my original intent for this column), doing it that way is difficult. We will follow an easier course by changing part families first and then upgrading to the latest version of *Quartus*. If the complexity of this method frustrates you, try to remember that this is the *easy* way. To paraphrase a common saying: "There are only two ways to do this. If you don't like this one, you for sure won't like the other one!" Here is the flow of part numbers and *Quartus* versions that we will use:

3C25 with v13.1 → 5CEFA7 with v13.1

→ 5CEFA7 with v15.0 → 5CEFA9 with v15.0

Notice that *Quartus II* version 13.1 does not support our 5CEFA9F23C8 part, so we pick a dummy part (5CEFA7F23C8) that it does support just to get us into the Cyclone V family. After we migrate to *Quartus II* version 15.0, we will pick our final, correct 5CEFA9F23C8 target. Also, notice that in the flow above, we only change *one* item in each step: either the part number or the *Quartus* version, but never both.

FPGA Code Porting Tasks

Now that we understand the mess that we have gotten ourselves into, here is an outline of the WA2DFI 6-step program to successful FPGA code porting:

1) Open the design in the original *Quartus* version.

2) Update the wizard-generated modules.

3) Add code to hook in new signals and remove unused old signals.

4) Add new location properties.

5) Update the SDC timing constraints file with new signals, and remove old signals.

6) Compile-debug-repeat.

While none of these steps is fraught with peril, some are a bit more involved than others. Let's look at each step in more detail.

Open the Design

To get a copy of the FPGA source code, download a copy of the *Quartus* archive from the SDRstick SVN webserver.[8] The archive not only contains the source files (with a .v extension), but the pin assignment file (.qsf extension), timing constraints file (.sdc extension) and many other files needed to successfully compile the complete project.

Once you have downloaded the archive file, start the *Quartus II* version 13.1 software and click on **<file><open project…>**. Navigate to the .qar file that you downloaded and click on it. From the dialog box that opens, select the destination folder (usually the default is good) and click **OK**. *Quartus* will extract all of the files from the archive and set up the project, all ready to go. I recommend that you fire off a trial compile now (yes, right now!) with no changes. This will tell you if you have everything set up correctly. The compile button is the small right-facing triangle on the toolbar. If you prefer menus, the **<Start Compilation>** button is also under the **<Processing>** menu. You should get a bunch of warnings from *Quartus*, but no errors. If *Quartus* reports errors, you must fix them before you can continue.

Now that we have a good compile, we need to do a little project clean up. By project, I am referring to the group of files that comprise the entire design. The Hermes design has changed and evolved over time. Some functions were removed or superseded by new and improved ones. Other pieces of code were rewritten to be more efficient. The net result is that there are files included in the project that are unused. Since we don't need to update unused modules, it is best to remove them now. There are about two dozen unused files that you can remove. I have listed them in a text file that you can download.[9]

First, remove the files on the list from the project directory or subdirectory. If you are cautious, like I am, create a new directory outside of the *Quartus* project and move the files there. That way *Quartus* will not be able to find them, but if you make a mistake and remove a needed file, you can easily restore it. Next, in *Quartus*, under the **<Project><Add/Remove Files in Project>** menu, remove the files from the project. You might think that deleting (or moving) the file is sufficient, but *Quartus* keeps track of the files that it *knows* are in the project. You must remove these or *Quartus* will look for them (in vain, since you moved them) and not be happy about not finding them. After you remove all of the dunsel files, make sure to click **<apply>** and **<OK>**.[10]

Check the **Files** tab of the **Project Navigator** window to see a list of files in the project. See Figure 1. You should recompile the project to make sure that you did not accidentally remove something that is necessary. Before you do this, however, you need to remove the intermediate database files for past compiles. This will ensure that all traces of the files that you removed are gone from *Quartus* "memory" of compiles past. Go into the project directory and remove the two directories **db** and **incremental_db** along with their contents. Don't worry; *Quartus* will re-create them as soon as you run a compile, which you should now do. As before, *Quartus* should report some warnings, but no errors.

Update Wizard-Generated Modules

The Altera MegaWizard Plug-In Manager was used to generate some of the modules in the Hermes code. The wizard, as I call it, is software built into *Quartus* that helps you set parameters for Altera functions such as FIFO, RAM and ROM memories, phase-locked loops (PLLs) and other functions. The Hermes design uses four PLLs, four FIFO memories, three ROM memories, one RAM memory and one multiplier for a total of 13 Wizard generated modules. Each of these modules must be updated first to the Cyclone V family under *Quartus II* version 13.1 before we can open them in *Quartus II* version 15.0.

Let's now move our design to the Cyclone V family. With the design open, select **<Assignments><Device>** from the menu bar. Select **Cyclone V** in the **Family** field. A dialog box will appear asking if you want to remove all location assignments. This tells *Quartus* to remove the old pin assignments that will no longer be valid when we change to a different part. This is important, since the Cyclone III pin numbers have

Altera Part Numbers Explained, Sort Of

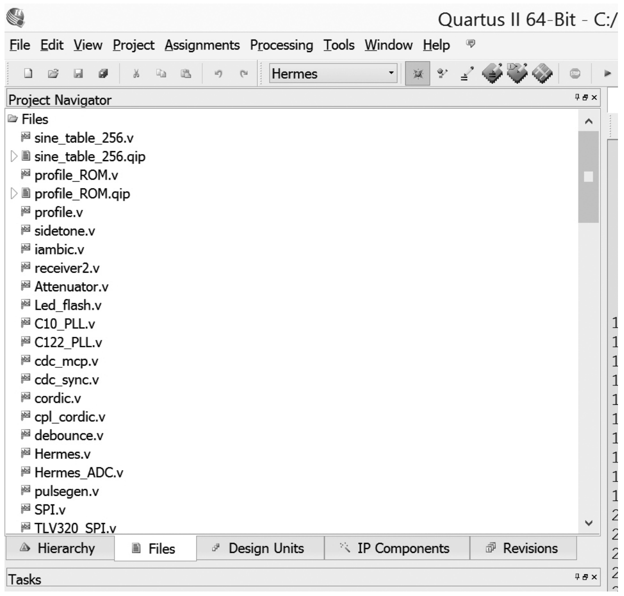
Just in case you are wondering what all those numbers mean in that long and involved FPGA part number, look no further. Our FPGA part number can be broken into 9 sections:

5C E F A9 F 23 C 8 N

The ***5C*** signifies that our part is in Altera's Cyclone V family of parts. Examples of other Altera part families are Stratix 5 (5S) and Arria 10 (10A). The ***E*** in our part number signifies Enhanced logic/memory, in other words, no embedded *hard-processor* or *high-speed transceivers* (the digital logic kind of transceiver, not the Amateur Radio variety). The ***F*** signifies that we have a *hard memory controller*, which is a DDR memory controller pre-built for us in silicon so we do not have to design one out of the FPGA fabric ourselves. The A9 tells us that this is the largest device in the family, with 301K Logic Elements (LEs). In contrast, the smallest member of the family, the A2, has only 25K LEs.

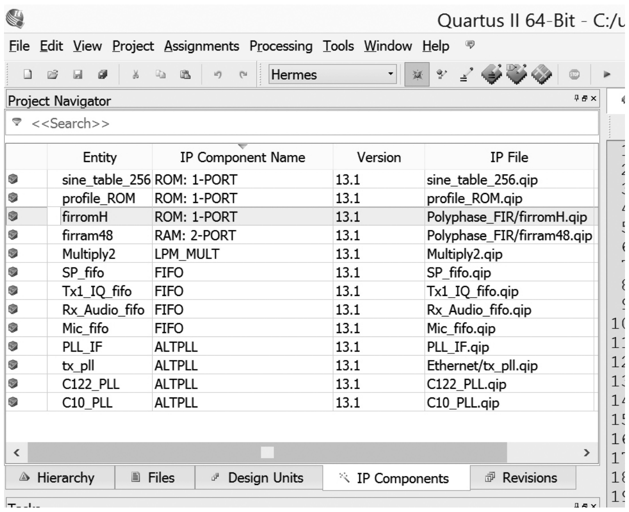
Moving along, ***F23*** represents the package type. F stands for Fine Line Ball Grid Array and 23 stands for the square package side dimension, 23mm. This package has 484 connections, each consisting of a solder ball on the bottom of the chip. The solder balls are arranged in a 22mm by 22mm square grid on 1mm centers. Don't try to mount this part with your American Beauty soldering iron![15]

The ***C*** stands for commercial temperature range (0°C to 85°C); there are two wider temperature ranges if needed. The ***8*** represents the speed grade. There are only three grades, 6 being the fastest (and most expensive). The 8 graded parts are the slowest (and cheapest), but still plenty fast enough for our application. As you would expect, grade 7 parts are in between 6 and 8 in performance. Last but not least, the N indicates lead-free packaging. No Ethyl for us, thank you.[16] More information than you ever wanted to know is available in the reference.[17]



Files
sine_table_256.v
sine_table_256.qip
profile_ROM.v
profile_ROM.qip
profile.v
sidetone.v
iambic.v
receiver2.v
Attenuator.v
Led_flash.v
C10_PLL.v
C122_PLL.v
cdc_mcp.v
cdc_sync.v
cordic.v
cpl_cordic.v
debounce.v
Hermes.v
Hermes_ADC.v
pulsegen.v
SPI.v
TLV320_SPI.v

Figure 1 — Project Navigator view of files in the project.



Entity	IP Component Name	Version	IP File
sine_table_256	ROM: 1-PORT	13.1	sine_table_256.qip
profile_ROM	ROM: 1-PORT	13.1	profile_ROM.qip
firromH	ROM: 1-PORT	13.1	Polyphase_FIR/firromH.qip
firram48	RAM: 2-PORT	13.1	Polyphase_FIR/firram48.qip
Multiply2	LPM_MULT	13.1	Multiply2.qip
SP_fifo	FIFO	13.1	SP_fifo.qip
Tx1_IQ_fifo	FIFO	13.1	Tx1_IQ_fifo.qip
Rx_Audio_fifo	FIFO	13.1	Rx_Audio_fifo.qip
Mic_fifo	FIFO	13.1	Mic_fifo.qip
PLL_IF	ALTPLL	13.1	PLL_IF.qip
tx_pll	ALTPLL	13.1	Ethernet/tx_pll.qip
C122_PLL	ALTPLL	13.1	C122_PLL.qip
C10_PLL	ALTPLL	13.1	C10_PLL.qip

Figure 2 — Project Navigator view of IP components in the project.

about the same chance of being the same as the Cyclone V pin numbers as my dog has of becoming President. (My cat agrees with me on this one.) This is especially true since the packages (QFP240 versus FBGA484) are completely different. So click **Yes** to remove them. To narrow your choices, select **FBGA** in the **Package** field, **484** in the **Pin count** field and **8** in the **Speed grade** field. Now select **5CEFA7F23C8** under Available Devices with a single click. Note that you will again have to confirm that you want to remove all location assignments, even though they have already been removed! Click **Yes** and then **OK**. That's it! You are now are using a Cyclone V part! Well, not the right part, and we are still using *Quartus II* version 13.1. We will fix both of these problems after we finish updating the wizard-generated modules.

To update the wizard-generated modules, we will use (are you ready for this?) the wizard itself! We will open each module in turn and tell the wizard to use the Cyclone V family and regenerate the module. This will work for all of the modules except the four PLLs and the ROM memory. We will handle them separately. To get started, click the **IP Components** tab of the **Project Navigator.** See Figure 2. You should see thirteen lines in the window. Leave the PLLs alone for now (**PLL_IF**, **tx_pll**, **C122_PLL** and **C10_PLL**) as well as the **firromH** module. Open **sine_table_256** by double clicking on it. The MegaWizard Plug-In Manager will start. In the upper right corner, the **Currently selected device family** will be **Cyclone III** and the **Match project/default** box will be checked. Uncheck this box and then select **Cyclone V** from the **Currently selected device family** drop-down menu. Click Finish twice and the wizard will update the module for you. Now repeat the same steps for the other 7 modules (profileROM, SP_fifo, firram48, Tx1_IQ_fifo, Rx_Audio_fifo, Multiply2 and Mic_fifo).

The firromH module must be handled differently, mainly because the designers broke one of the rules and modified the firromH.v file that the wizard created. There are reasons why they did this (which I am not going to elaborate on), but the consequences are that the new wizard-generated firromH.v file will over-write the modified old version. We will have to re-modify the new file with the old changes to make it work. Go ahead and open the firromH module in the wizard and convert it to the Cyclone V family just like you did with the other modules. After you do this, click on the **Mem Init** tab in the toolbar. We must specify an existing file name in order to satisfy the wizard, so click on the **Browse** button and select the **Polyphase_FIR** directory and pick any of the files that end in **.mif**. You will have to change the selection to **MIF files** in the drop-down **Files of type** box at the bottom of the window to make the .mif files visible. It does not matter which file you choose, since we are going to manually change it in the firromH.v file in the next step.

Now we are going to modify the firromH.v file that the wizard created for us. (Shh, don't tell the wizard!) In the *Quartus* Project Navigator pane, click on the **Files** tab, find the **firromH.v** file (hint: it is called "Polyphase_FIR/firromH.v" because it is in a project sub-directory) and open it by double-clicking on it. After line 43, add line 44:

parameter MifFile = "missing_file.mif";

Follow this with a blank line 45 to make things readable. Next go to line 88 and change it to read:

altsyncram_component.init_file = MifFile,

Make sure to type it exactly as shown, since capitalization and punctuation matter. Save it and close the file. We will delve more into why we made this change in the next column, when we dig into the code. Right now we need to finish up with the wizard by updating the PLL modules.

Unfortunately, Cyclone V PLLs are different from Cyclone III PLLs, so we cannot just upgrade them using the wizard. We must create new ones and replace the old ones with the new ones. First, remove the old PLL_IF, C10_PLL, C122_PLL and tx_pll files from the project using the Add/Remove Files in Project menu. Each of these modules will have several files (typically .v and .qip files); make sure that you remove all of them. Next remove the files from the project directory and sub-directories. (The **tx_pll** files are in the Ethernet sub-directory.) While you are at it, remove the **db** and **incremental-db** directories and their contents, just like you did before.

To create a new PLL module, open the wizard using **<Tools><MegaWizard Plug-In Manager>** and select **Create a new custom megafunction** from the list. From the list of functions, pick **Altera PLL v13.1** from the PLL submenu. In the output file box append the name (for example, **PLL_IF_new**) after the string that represents the project directory. This will name your module and place it in the project directory. All four

Table 1
Wizard Settings for New PLL Modules

	PLL_IF_new	*tx_pll_new*	*C10_PLL_new*	*C122_PLL_new*
Reference Clock	122.88 MHz	50.0 MHz	10.0 MHz	122.88 MHz
Number of clocks	4	4	2	2
Multiply Factor (M)	4	10	64	9
Divide Factor (N)	1	1	2	3
outclk0 cascade?	N	N	Y	Y
outclk0 Divide Factor (C)	40	4	32	192
outclk0 output	12.288 MHz	125 MHz	n/a	n/a
outclk0 phase shift	0°	0°	0°	0°
outclk1 cascade?	N	N	N	N
outclk1 Divide Factor (C)	160	4	125	24
outclk1 output	3.072 MHz	125 MHz	80 kHz	80 kHz
outclk1 phase shift	0°	90°	0°	0°
outclk2 cascade?	Y	N	n/a	n/a
outclk2 Divide Factor (C)	256	40	n/a	n/a
outclk2 output	n/a	12.5 MHz	n/a	n/a
outclk3 cascade?	N	N	n/a	n/a
outclk3 Divide Factor (C)	40	200	n/a	n/a
outclk3 output	48 kHz	2.5 MHz	n/a	n/a

PLL modules have these common settings:

- Device Speed Grade: 8
- PLL Mode: Integer-N PLL
- Operation Mode: direct
- Enable locked output port: checked
- Enable physical output clock parameters: checked

Set the other parameters for each module to what I have listed in Table 1. Leave all other parameters set to their default settings. When you click **Finish** and **Exit** after specifying all the parameters, *Quartus* will ask you if you want to add the new IP to the project. Click **Yes**. Since the PLL modules need to be added to the project eventually, this will save you a step later.

The last step is to open the source file that instantiates each PLL, update the module name and check (and correct, if necessary) the module connections. The **tx_pll** is used in the **rgmii_send.v** file in the Ethernet subdirectory. The other three are instantiated in the top level **Hermes.v** file. I will guide you through the first one, and you can follow the same procedure on the other three on your own. (You didn't think I was going to do *all* of it for you, did you?) Open the top level **Hermes.v** file and also the **C122_PLL_new.v** file. Go to line 1385 in **Hermes.v** and you will see the instantiation of C122_PLL. The instantiated name is PLL_inst, and the port names are inclk0, c0 and locked. Observe that port **inclk0** is connected to **_122MHz**, port **c0** is connected to **osc80khz** and port **locked** is not connected to anything. Now look at the **C122_PLL_new.v** file. You will see that there are now four ports instead of three: **inclk0** is now called **refclk**, **c0** is now called **outclk_1** and **locked** remains unchanged. The new input is called **rst;** we will not use it. Change line 1385 to read:

```
C122_PLL_new PLL_inst (.refclk(_122MHz), .outclk_1(osc_80khz), .locked( ), .rst( ) );
```

The C10_PLL_new and PLL_IF_new modules will require similar changes.

The astute reader will notice that the wizard allows you to turn off the **locked** output when it is unused, but the new PLLs all have an **rst** input that cannot be disabled. Ideally this input should be connected to reset logic; however, we will save code improvements for a later time. I need to call your attention to one other change that I slipped in while you were not looking. I changed the reference clock of the **tx_pll** module from 125 MHz to 50 MHz. I did this out of necessity, since the CVA9 does not have a 125 MHz clock input! It does have a 50 MHz clock input, but since we have not added it to the top-level **Hermes.v** source file yet, just leave it at 125 MHz. We will fix it shortly.

Now that you are experienced in matching up old port names to new port names, this would be a good time to go back and check the other nine wizard-generated modules that we updated to make sure that the port definitions in each module's .v file match up with the ports called out at the module's instantiation. Here's a quick hint: all of them are instantiated in **Hermes.v** except for sine_table_256 (sidetone.v), Multiply2 (sidetone.v), profile_ROM (profile.v) firromH (Polyphase_FIR/firx2r2.v) and firram48 (Polyphase_FIR/firx2r2.v).

As a short aside, I keep a note pad handy to write down things like "change 125M clock to 50M" as a note to myself. When you are updating the code, you will likely perform many tasks out of order and it is easy to forget a simple change that you queued up in your memory and then forgot about it.

At this point, we are finished with *Quartus II* version 13.1. Close *Quartus* and re-open the project in *Quartus II* version 15.0. The new version of *Quartus* will ask you if it should overwrite the database with the new format. You can safely answer **Yes**. Change the part number to 5CEFA9F23C8 and run a compile to see if we broke anything. Now we are using version 15.0 with the correct FPGA part number. The light at the end of the tunnel is coming into view, and it is not an oncoming train!

Add and Remove Code and Signals

The next step in our 6-step program is to match up the old design (Hermes) signals with the new design (CVA9) signals. We must account for *every one* of the Hermes signals, whether it is to remove it, change it to match the new CVA9 hardware or just connect it to its counterpart in the new design. We must also account for *every one* of the new design pins (CVA9) by either ignoring it, adding code to support it or just connecting it to its counterpart from the old design. In order to be able to do all of this cross checking, we need to map the old pin names (in this case from the Hermes board) to our new pins on the BeMicroCVA9 board. Some of these signals connect to parts on the BeMicroCVA9, and some connect directly to the HF2 and TX2 boards that are plugged into the BeMicroCVA9.

What we need is a table that shows the old name alongside the new name and the new FPGA pin number. (We will use the pin numbers in the next section.) You could figure this out for yourself, but I have created a file for you containing a table of all of the signal names in the design to give you a head start. This **Hermes_6_to_IQ2_pins** table will tell us which pins map directly onto new pins and which do not.[11] An excerpt of this table (showing only the signals that we need to change) is shown in Table 2. All of the changes will be made to the top level **Hermes.v** file.

A quick look at the full table will reveal that most Hermes signals have equivalent (although differently named) BeMicroCVA9 signals. We can leave these alone. The other signals fall into three categories:

1) The Hermes signal has a different or shared function than the CVA9 signal.

2) The Hermes signal does not exist in the CVA9 design.

3) The CVA9 signal does not exist in the Hermes design.

In the first case, we must modify the

Hermes code to connect to the CVA9 hardware that is different from the Hermes hardware. As an alternative, we can choose to not implement the Hermes function on the different CVA9 hardware. This involves removing (typically by commenting out) code that connects to the removed pins. As we will explain next, you have to be careful when removing inputs.

In the second case, we can simply remove Hermes code that does not have CVA9 hardware associated with it. We must be careful to follow Hermes inputs all the way to their destinations and remove them cleanly. We do not want any floating inputs. There may be one or more required inputs to the Hermes code that came from hardware that does not exist on the CVA9. We will have to add new code to create these signals and set them to a valid state.

In the third case we must add code to the Hermes design to connect to the CVA9 hardware that does not exist in the Hermes design. As an alternative, we can choose to ignore the new hardware, but we must still drive any output pins to some known state to avoid hardware problems later.

Here are the index numbers (from the **Hermes_6_to_IQ2_pins** table) that belong to each category:

Category 1: 4, 5, 50, 51, 52, 53

Category 2: 28, 49, 67, 70, 73-76, 78-85, 91-104, 113-115

Category 3: 21, 22, 116-120

Category 1 Changes

These 7 pins all revolve around a hardware difference between the Hermes and the CVA9/HF2/TX2 hardware. Hermes has a 31 dB step RF attenuator and a single audio CODEC for receive audio output and microphone audio input. These two devices have separate serial interfaces (3-wire for the attenuator and 3-wire for the CODEC). The HF2 receiver has the same attenuator and CODEC (which is used for receive audio output only), but they share clock and data lines, each having a separate chip-select. This makes the interface 4 lines to both parts. To complicate things, the TX2 transmitter has another CODEC (used only for microphone audio input) that shares the same clock and data lines, but with its own separate chip select. So now the new five-line interface must communicate with three parts over common clock and data lines using three different chip-selects. Rather than devise logic to adapt the two Hermes ports to the special five-line CVA9/HF2/TX2 interface, I have opted to just disable the HF2 and TX2 CODECs by tying their chip-selects to the inactive state. The PowerSDR™ software can use the sound card in place of the CODECs, so this does not create a hardship. We can go back later and add the code in if we want to.

Table 2
Excerpt of the Hermes_6_Sept_to_IQ2_Pins File

Index	*Hermes name*	*Hermes FPGA pin*	*HF2 Name*	*TX2 Name*	*CVA9 J2 pin*	*CVA9 name*	*CVA9 FPGA pin*	*Description*
4	ATTN_DATA	39	SPI_DATA	SPI_DATA	62*	EG_P58	V20	Data Output To Attenuator
5	ATTN_CLK	22	SPI_CLK	SPI_CLK	64*	EG_P59	U20	Clock Output To Attenuator
21	INA14		INA14	-	41	EG_P17	AB20	Input Data From ADC
22	INA15		INA15	-	43	EG_P18	Y20	Input Data From ADC
28	SHDN	194	-	-	-	-	-	Shutdown Output to ADC
49	CMODE	230	-	-	-	-	-	Mode Select Output To CODEC (I2C or SPI)
50	nCS	231	PH_CODEC_nCS	-	60	EG_P57	V19	Chip Select Output To CODEC
51	nCS	231	-	MIC_CODEC_nCS	57	EG_P24	Y21	
52	MOSI	226	SPI_DATA	SPI_DATA	62*	EG_P58	V20	SPI Data Output To CODEC
53	SSCK	224	SPI_CLK	SPI_CLK	64*	EG_P59	U20	SPI Clock Output to CODEC
67	PHY_CLK125	149	-	-	-	-		125 MHz Clock Input From PHY PLL
70	CLK_25MHZ	33	-	-	-	-	-	25 MHz Clock Input From PHY oscillator
73	SCK	68	-	-	-	-	-	Clock Output To MAC EEPROM
74	SI	38	-	-	-	-	-	Data Output To MAC EEPROM SI pin
75	SO	70	-	-	-	-	-	Data Input From MAC EEPROM SO pin
76	CS	87	-	-	-	-	-	Chip Select Output To MAC EEPROM
77	NCONFIG	63	-	-	-	RECONF	G6	Reload FPGA From Config Prom When High
116	-	-	DRV_CLK_OUT_N-	-	67	EG_P29	U21	Output To HF2: Drive 122.88 MHz to J1 pin 5
117	-	-	-	DAC_CLK	3	RESET_EXPn	U13	Output To TX2: Clock To DAC
118	-	-	-	EN_RX_ANT	12	EG_P37	M7	Output To TX2: T/R Switch
119	-	-	-	-	-	DDR3_CLK_50MHZ	H13	Input From 50 MHz Oscillator
120	-	-	-	-	-	CLK_24MHZ	M9	Input From 24MHZ Oscillator

*shared pins

As they used to say in college, it is left as an exercise for the student.

We will leave the signals at index 4 (**ATTN_DATA**) and index 5 (**ATTN_CLK**) alone, which will allow normal control of the RF attenuator. We will remove the signals at index 50 and 51 (**nCS**), index 52 (**MOSI**) and index 53 (**SSCK**). Open **Hermes.v** and look at lines 154 to 156. Rather than delete the lines of code that we might want to add back in someday, just comment them out by adding two slashes (//) at the beginning of each line. To complete this change, we must also remove the signals that drove these outputs. Go to line 519 and delete the signals **nCS**, **MOSI** and **SSCK** from inside the parentheses. (Yes, this will leave an empty field between the parentheses. This is how you specify no connection.) This effectively disconnects the **.nCS**, **.MOSI** and **.SSCK** ports of the TLV320_SPI module from the top level outputs that no longer exist. While this is not a complete removal of the TLV320_SPI module, it is close enough; *Quartus* will remove the unused logic for us. We will still have the ability to easily connect it back up at a future date when we are ambitious enough to combine its outputs with the attenuator interface and make the CODECs work again.

Now that we have removed the signals for the Hermes CODEC, we must define and drive the two new CODEC chip selects to their inactive state. Since they are active-low, we will drive them high. Add these two lines right after the SSCK port definition that you just commented out:

```
output PH_CODEC_nCS,
output MIC_CODEC_nCS,
```

Insert the following lines of code in a convenient place. Right after the module definition around line 237 is a good place:

```
assign PH_CODEC_nCS = 1'b1;
assign MIC_CODEC_nCS = 1'b1;
```

Category 2 Changes

These are perhaps the easiest changes to make. Inputs are handled differently than outputs. Outputs are handled as above: comment out the output pin and remove (or comment out) the source of the signal. Simply search for each signal name in turn and comment out its definition and its source. Inputs must be tied to a known (typically inactive) state after the input pin definition is commented out. We must also define an internal pin to replace the input pin definition that we commented out. First, let's identify the inputs from the list of Category 2 changes listed above. They are index numbers 67, 70, 75, 80, 91, 92, and 94 to 97. Find each of them in the full table, locate the corresponding input pin definitions in **Hermes.v** and comment them out. Note that the inputs CLK_25MHZ, ANT_TUNE, IO2, IO4, IO5, IO6 and IO8 are unused in the code, so they require no further changes. The signals SI and ADCMISO do need to be set to a known state. To do this, insert the following lines of code in a convenient place. Right after the Category 1 lines you added above is a good place:

```
wire SO;
assign SO = 1'b0;
wire ADCMISO;
assign ADCMISO = 1'b0;
```

The last input we need to handle is special: the PHY_CLK125 clock input. Remember from our scratchpad memo notes that this clock does not exist in the CVA9. We have already changed the tx_pll module to use a 50 MHz clock, which we will now add and connect up in place of the missing 125 MHz clock. Add the following code after line 166 (just below the **input PHY_CLK125** line that you commented out:

```
input DDR3_CLK_50MHZ,
```

Now search for PHY_CLK125 (ctrl-F opens a find window in *Quartus*) and change it to **DDR3_CLK_50MHZ** in two places: within the parentheses in the network module instantiation (around line 400) and near the end of the file in the always block heartbeat LED definition. Yes, it will make the heartbeat LED flash a bit slower, but that is acceptable.

To remove the outputs, comment out each output line in the module pin definitions at the beginning of the file (just like you did with the Category 1 outputs). In addition, remove the pin from inside the parentheses in a module instantiation (again, just like you did with the category 1 outputs) or comment out the assignment statement in which it appears. The signals **USEROUT0** – **USEROUT7** are a special case because they are assigned to the signals **Open_Collector[1]** – **Open_Collector[7]**, respectively. You must comment out the assignment (within the parentheses) of **Open_Collector** in the instantiation of the High_Priority_CC module (around line 1200) as well as the following assignment (around line 1144):

```
wire [7:0] Open_Collector;
```

Category 3 Changes

The index 21 and 22 changes widen the ADC data bus from the Hermes code 14 bit width to the HF2 receiver ADC width of 16 bits. We need to change the code in the **always** block that defines the variable **temp_ADC**. This variable is already 16 bits wide, but only the top 14 bits are connected to the 14 **INA** inputs from the ADC. Change line 135 to read:

```
input [15:0] INA,
```

Search for temp_ADC (around line 870) and change the **always** block to read as follows:

```
always @ (posedge C122_clk) begin
   temp_DACD        <= {DACD, 2'b00};
   if (RAND) begin
      if (INA[0])
        temp_ADC <= {~INA[15:1], INA[0]};
      else
        temp_ADC <= INA;
   end
   else
       temp_ADC <= INA;
end
```

For indexes 116 and 118 we need to add two new output signals and assign values to them. Index 117 is the output clock to the transmit DAC. On the Hermes board, the 122.88 MHz oscillator feeds the DAC directly without FPGA involvement. The TX2 transmitter DAC requires a clock from the FPGA, so we must add it. Finally, index 120 is an unused 24 MHZ oscillator input. Even though it is not currently used, we need to assign it to an input so we can fix the input pin location in the pin list. Add the following pin definitions to the Hermes module pin list at the beginning of the **Hermes.v** file:

```
output DRV_CLK_OUT_N,
output DAC_CLK,
output EN_RX_ANT,
input CLK_24MHZ,
```

If you insert these at the end of the pin list, remember that a comma separates each pin definition, and there is no comma after the last one. Now insert the following code in a convenient place (after your previous Category 2 code additions is a good place):

```
assign DRV_CLK_OUT_N = 1'b1;
assign DAC_CLK = _122MHz;
assign EN_RX_ANT = 1'b1;
```

Go ahead and compile again, just to make sure that you didn't forget a semicolon or make some other easy-to-fix syntax error.

Add New Location Properties

Do you remember those location properties that we removed when we changed part numbers? We now have to add them back into the design, except that we want to add the pin numbers for our new device in place of the old numbers that we removed. This is why I included the pin numbers in Table 2.

The best way to add new location properties to the design is to write a script file that contains a line for each new location assignment. The format for each line is:

set_location_assignment PIN_*xxxx* –to *signal_name*

where:

xxxx is the device pin number, and
signal_name is the pin name from the top

design file (**Hermes.v**).

Again, I have created a file for you to save you the effort of typing all those lines into the script file. You can download it from the SDRstick SVN webserver.[12]

To run the script, place the file in your top directory (that is, the directory that contains your **Hermes.qsf** file and all of your Verilog source files). Now add it to your project using **<Project> <Add/Remove Files in Project…>**. Under **<Tools> <Tcl Scripts…>**, select the file and click **Run**. All of your pin locations from the script file have now been added. If you want to check your new assignments (or maybe you just don't believe me) you can open the Assignment Editor from (where else) the **<Assignments> <Assignment Editor>** menu. You should see all of your new **Location** assignments listed. Run another compile to make sure things are as they should be.

Update SDC Timing Constraints

The last task we must undertake is to review the **Hermes.sdc** timing constraints file line by line to remove constraints for signals that we have removed, add (or expand existing) constraints for new signals and update constraints for anything that we changed. I will cover this in my next column. In the meantime, take a look at the file to become familiar with it. Did I just give you a homework assignment? Sorry.

Compile-Debug-Repeat

The focus of our efforts has been on obtaining a good compile of our code under the new *Quartus* version while targeting the new FPGA part. That said, a successful compile does not necessarily mean we have a working design. Now is the time to review all of those *Quartus* warnings that we have so blithely been ignoring all this time. Most (if not all) of them can be ignored, but we must make sure of this. The cause of the ones that cannot be ignored must be fixed. The final step, of course, is to load the compiled programming file into the BeMicroCVA9 and test it to make sure that it works. I will cover review of warnings and how to fix them, timing constraints update and how to load and run the code on real hardware in my next column. An updated *Quartus* archive containing all of the changes that we have made is available on the SDRstick SVN webserver.[13]

Table 3
Files

Quartus archive of original source code:	Hermes_6_Sept.qar
Table of pin cross references:	Hermes_6_Sept_to_IQ2_pins.pdf
List of unused files:	Hermes_6_Sept_unused_files.txt
Tcl script to reassign location properties:	Hermes_6_Sept_map_pins.tcl
Quartus archive of ported source code:	Hermes_6_Sept_ported.qar

What's Next?

Now that you know how to port FPGA code to new devices, what can you do with this skill? The openHPSDR project is open source, and the Apache Labs Anan series of transceivers are all powered by open-source FPGA firmware. The FPGA code to implement any or all of the features of these transceivers is available for your use. When a new feature comes out, you can look at how it is done and integrate that function into your radio. Better yet, you can add your own feature and show everyone else how to improve their own rigs. That is the true benefit of open-source!

Each openHPSDR board has an on-board FPGA and Verilog code to match. All of it is available from the openHPSDR repository, and you are now qualified to port it to any new hardware that you can scrounge up.[14] The tools that you have used today are the very same tools that the developers use when they write or update the code.

As always, please drop me an e-mail if you have any suggestions for topics you would like to see covered in future Hands-On-SDR columns or even just to let me know whether or not you found this discussion useful.

Notes

[1]Scotty Cowling, WA2DFI, "Hands On SDR," *QEX*, Mar/Apr 2015, pp 9-19.
[2]The BeMicroCVA9 board is available from from Arrow Electronics: **parts.arrow.com/item/detail/arrow-development-tools/bemicrocva9**.
[3]The UDPSDR-HF2 receiver board is available from Arrow Electronics: **parts.arrow.com/item/detail/arrow-development-tools/udpsdr-hf2.**
[4]The UDPSDR-TX2 transmitter board is available from Arrow Electronics: **parts.arrow.com/item/detail/arrow-development-tools/udpsdr-tx2**.
[5]The UDPSDR-HF1 receiver board is available from Arrow Electronics: **parts.arrow.com/item/detail/arrow-development-tools/udpsdr-hf1**.
[6]For more information about Hermes-Lite see: **github.com/softerhardware/Hermes-Lite/wiki**.
[7]To download the free Altera Web Edition software, go to: **dl.altera.com/?edition=web**.
[8]The source code is available from the SDRstick SVN at: **svn.sdrstick.com** under the <**sdrstick-release/BeMicroCV-A9/Hermes-HF2-Port/firmware/source**> directory. The file name is <**Hermes_6_Sept.qar**>. It is also available for download from the ARRL *QEX* files web page. Go to **www.arrl.org/qexfiles** and look for the file **1x16_Cowling_Hands_On_SDR.zip**.
[9]The list of unused files is available from the SDRstick SVN in the same directory as listed in Note 8. The file name is <**Hermes_6_Sept_unused_files.txt**>. This file is also part of the **1x16_Cowling_Hands_On_SDR.zip** file, also as listed in Note 8.
[10]dunsel, noun, (slang, from Star Trek) a part that serves no useful purpose.
[11]The cross reference of Hermes to IQ2 pins is available from the SDRstick SVN in the same directory as given in Note 8. The file name is <**Hermes_6_Sept_to_IQ2_pins.xls**>. The file is also included in the **1x16_Cowling_Hands_On_SDR.zip** as given in Note 8.
[12]The pin location Tcl script file is available from the SDRstick SVN in the same directory as given in Note 8. The file name is <**Hermes_6_Sept_map_pins.tcl**>. The file is also included in the **1x16_Cowling_Hands_On_SDR.zip** file.
[13]Source code containing all of the changes outlined in this column is available from the SDRstick SVN at **svn.sdrstick.com** under the <**sdrstick-release/BeMicroCV-A9/Hermes-HF2-Port/firmware/source**> directory. The file name is <**Hermes_6_Sept_ported.qar**>. This file is also included in the ZIP file, as listed in Note 8.
[14]For HPSDR firmware, look in the TAPR repository, **svn.tapr.org** in <**main/trunk**> under the board name.
[15]American Beauty soldering irons of old were massive and the larger ones could solder copper pipes. Like the term "boat anchor," the term is an affectionate name for a tool of the past. Lo and behold, they are still in business! I especially like the handheld unit shown at **americanbeautytools.com/Soldering-Irons/19/features**.
[16]An obscure and wholly unwarranted reference to tetraethyllead $(CH_3CH_2)_4Pb$, an octane booster added to gasoline from about 1920 until the early 1990s in the USA. Premium gasoline was referred to as "Ethyl" to us old timers.
[17]More information on part numbers can be found in Altera's Cyclone V Device Overview at **altera.com/en_US/pdfs/literature/hb/cyclone-v/cv_51001.pdf**.

2015 *QEX* Index

We Design And Manufacture To Meet Your Requirements
*Protoype or Production Quantities

800-522-2253

This Number May Not Save Your Life...

But it could make it a lot easier! Especially when it comes to ordering non-standard connectors.

RF/MICROWAVE CONNECTORS, CABLES AND ASSEMBLIES

- Specials our specialty. Virtually any SMA, N, TNC, HN, LC, RP, BNC, SMB, or SMC delivered in 2-4 weeks.
- Cross reference library to all major manufacturers.
- Experts in supplying "hard to get" RF connectors.
- Our adapters can satisfy virtually any combination of requirements between series.
- Extensive inventory of passive RF/Microwave components including attenuators, terminations and dividers.
- No minimum order.

NEMAL
Cable & Connectors for the Electronics Industry

NEMAL ELECTRONICS INTERNATIONAL, INC.
12240 N.E. 14TH AVENUE
NORTH MIAMI, FL 33161
TEL: 305-899-0900 • FAX: 305-895-8178
E-MAIL: INFO@NEMAL.COM
BRASIL: (011) 5535-2368

URL: WWW.NEMAL.COM

Down East Microwave Inc.

We are your #1 source for 50MHz to 10GHz components, kits and assemblies for all your amateur radio and Satellite projects.

Transverters & Down Converters, Linear power amplifiers, Low Noise preamps, coaxial components, hybrid power modules, relays, GaAsFET, PHEMT's, & FET's, MMIC's, mixers, chip components, and other hard to find items for small signal and low noise applications.

We can interface our transverters with most radios.

Please call, write or see our web site **www.downeastmicrowave.com** for our Catalog, detailed Product descriptions and interfacing details.

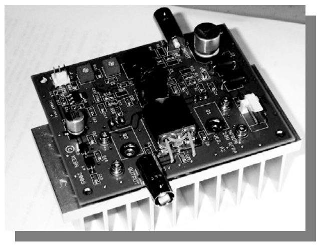
Down East Microwave Inc.
19519 78th Terrace
Live Oak, FL 32060 USA
Tel. (386) 364-5529

TAPR + HPSDR

HPSDR is an open source hardware and software project intended to be a "next generation" Software Defined Radio (SDR). It is being designed and developed by a group of enthusiasts with representation from interested experimenters worldwide. The group hosts a web page, e-mail reflector, and a comprehensive Wiki. Visit www.openhpsdr.org for more information.

TAPR is a non-profit amateur radio organization that develops new communications technology, provides useful/affordable hardware, and promotes the advancement of the amateur art through publications, meetings, and standards. Membership includes an e-subscription to the *TAPR Packet Status Register* quarterly newsletter, which provides up-to-date news and user/technical information. Annual membership costs $25 worldwide. Visit www.tapr.org for more information.

NEW!



PENNYWHISTLE
20W HF/6M POWER AMPLIFIER KIT

TAPR is proud to support the HPSDR project. TAPR offers five HPSDR kits and three fully assembled HPSDR boards. The assembled boards use SMT and are manufactured in quantity by machine. They are individually tested by TAPR volunteers to keep costs as low as possible. A completely assembled and tested board from TAPR costs about the same as what a kit of parts and a bare board would cost in single unit quantities.

HPSDR Kits and Boards

- **ATLAS** Backplane kit
- **LPU** Power supply kit
- **MAGISTER** USB 2.0 interface
- **JANUS** A/D - D/A converter
- **MERCURY** Direct sampling receiver
- **PENNYWHISTLE** 20W HF/6M PA kit
- **EXCALIBUR** Frequency reference kit
- **PANDORA** HPSDR enclosure

TAPR

PO BOX 852754 • Richardson, Texas • 75085-2754
Office: (972) 671-8277 • e-mail: taproffice@tapr.org
Internet: www.tapr.org • Non-Profit Research and Development Corporation

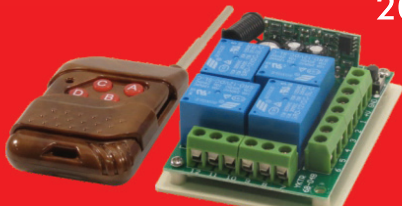
Quicksilver Radio

Test Equipment

USB Microscope

Up to 500X magnification. Captures still images and records live video. Built in LED Lighting. A must for working on suface mount components.

Wireless Relay Switch



200′+ Range. We have single, four, and eight channel models.

GO-PWR Plus™

Portable power to go or backup in the shack. Includes Powerpoles, bright easy to read meter, and lighted switch. For U1 size (35 ah) and group 24 (80 ah) batteries.

Digital Voltmeter/ Ammeter

Two line display shows both current and voltage. Included shunt allows measurement up to 50A and 99V. Snaps into a panel to give your project a professional finish.

LCR and Impedance Meter

Newest Model.
Analyzes coils, capacitors, and resistors. Indicates complex impedance and more.

Automatic Passive Component Analyzer

Analyzes coils, capacitors, and resistors.

Advanced Semiconductor Component Analyzer

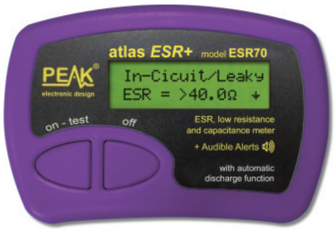
Analyzes transistors, MOSFETs, JFETs, IGBTs, and more. Graphic display. Enhanced functionality with included PC software.

Semiconductor Component Analyzer

atlas DCA
model DCA55
PEAK electronic design
NPN Silicon
Transistor
on - test
scroll - off
semiconductor component analyser
automatic lead identification
warning: avoid charge/voltages

Analyzes transistors, MOSFETs, JFETs and more. Automatically determines component pinout.

Capacitance and ESR Meter



Analyzes capacitors, measures ESR.

Get All Your Ham Shack Essentials at Quicksilver Radio Products. Safe and Secure Ordering at:

www.qsradio.com