Open Hardware Journal

February 2012 – **Open-Access Journal**, free to read, copy, and redistribute.

Contents

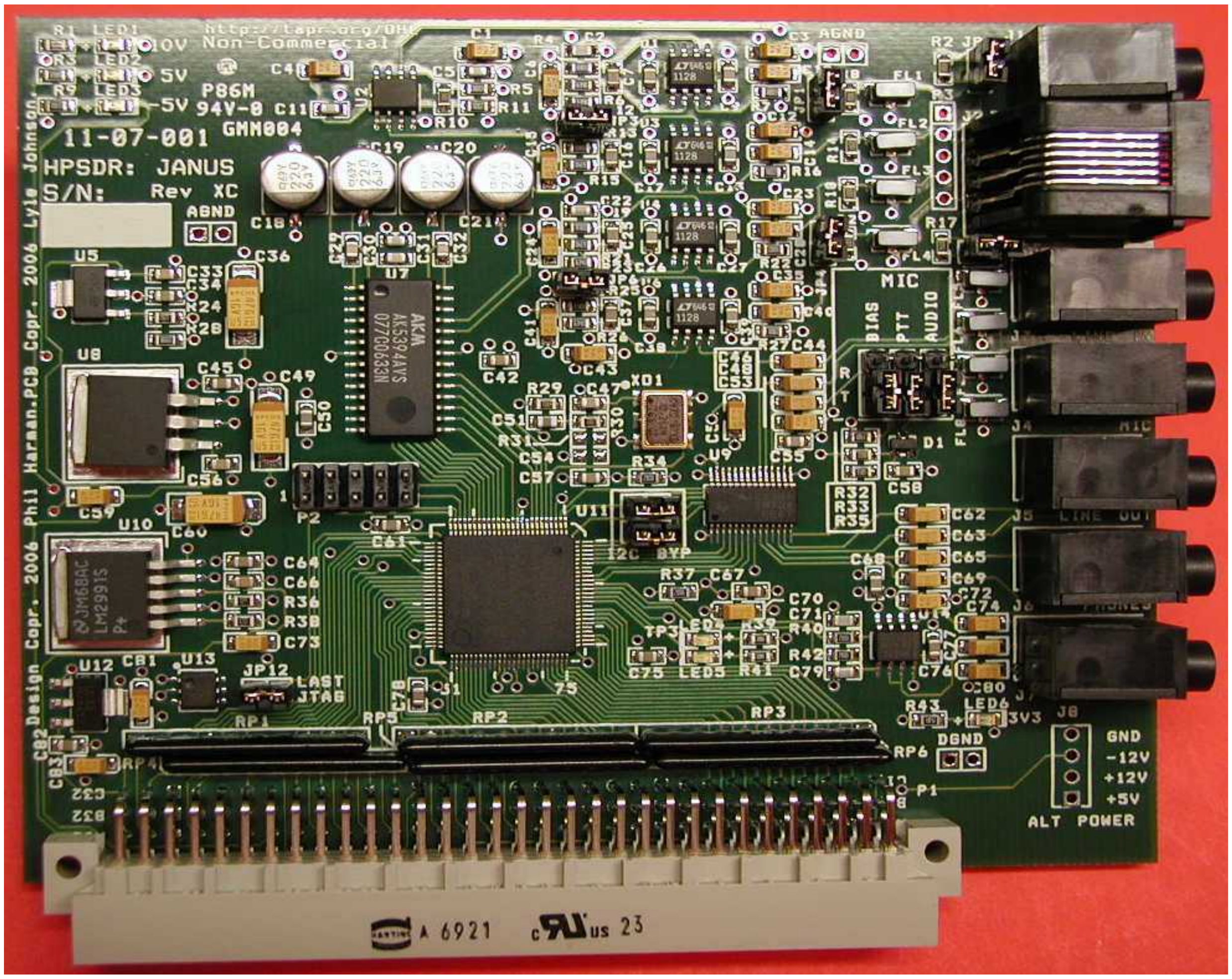


Photo 1: TAPR's HPSDR Janus board, a high-performance ADC/DAC for baseband radio, is a component in the HPSDR software-defined radio chassis. The board is now under the TAPR Open Hardware License.

Open Hardware Journal

Published by the *Open Hardware* organization. Please see our web site at OpenHardware.org

Editor: Bruce Perens <bruce@perens.com>

***Open Hardware* means sharing the design of physical or electronic objects with the public, similarly to Open Source software. The right to use, modify, redistribute, and manufacture, commercially or as a non-profit, is granted to everyone without any royalty or fee. Thus, Open Hardware designers hope to enrich society by developing a library of designs for useful objects that everyone can make, use, and improve.**

Editorial

This issue of Open Hardware Journal has come to release two months late, due to pressure on my time. The next issue will debut on May 1. It sounds like Quarterly is a good start, for now.

Our first issue was very well received, with about 10,000 downloads. The main negative comment is about our typography and layout. LibreOffice is our chosen tool, but it's a word-processor rather than a prepress and page-layout application. It's chosen simply because it lets us put together the issue without eating too much precious volunteer time, and it's Open Source.

Another request has been for an ePub version of the journal, for tablets. We're trying.

High Performance Software Defined Radio - An Open Source Design

By Scotty Cowling, WA2DFI <scotty@tonks.com>

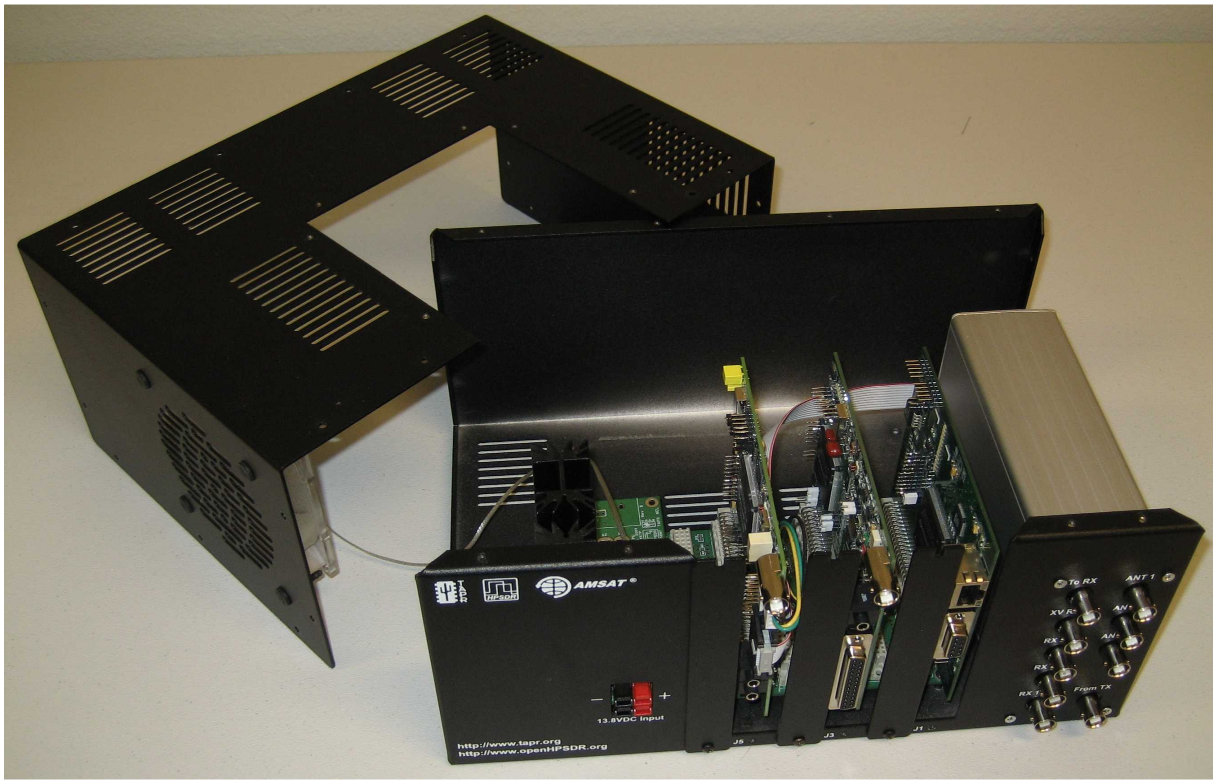


Photo 2. openHPSDR Transmitter/Receiver

From left to right, LPU, Mercury receiver, Pennylane transmitter, and Metis Ethernet interface are plugged into Atlas backplane. Alex filters (in aluminum enclosure) are on the right. All boards fit within the Pandora enclosure.

Introduction and History

Since its inception in 2005, the High Performance Software Defined Radio project has produced over a dozen building blocks that can be used to assemble a high-grade 100kHz to 55MHz software-defined radio (see Photo 2).

The openHPSDR project, as it is known today, began in March 2006 from the merger of the HPSDR Yahoo group and the Xylo-SDR e-mail reflector. The first piece of hardware produced was the Atlas backplane. Eric Ellison, AA4SW, paid for the initial run of 400 PCBs and shipped them to individuals from his dining room table in May of 2006. He collected enough money (entirely on the honor system) from these early adopters to pay for the initial PCB run. Due mainly to Eric's efforts, TAPR got involved on the production side in June 2006 and was able to help augment the many HPSDR designers' efforts with early volume production and storefront retail sales. While TAPR offers financial support to the designers to help defray some or all of the costs of building prototypes for testing, TAPR and the HPSDR project always were and remain independent entities. The HPSDR project changed its name to openHPSDR in April 2009 in order to more accurately reflect the open-source nature of the project.

In fact, the openHPSDR project was the impetus for creating the TAPR Open Hardware License ("OHL"). The openHPSDR developers wanted to create a community around their designs, much like the GNU General Public License, and invited TAPR to work with them to develop a license for open hardware designs. The OHL itself was the result of an open design process that included a public comment period. It was released in May, 2007, and is available for use by any open hardware project. Here is a link to the OHL: <http://www.tapr.org/ohl>

TAPR <http://tapr.org/> is a non-profit corporation that provides resources for the purpose of advancing the state of the radio art, especially the *digital* radio art. What could be more digital than a software defined radio with an A-to-D conversion practically at the antenna?

The openHPSDR project <http://openhpsdr.org/> is a community (currently over 1000 strong) of designers, developers and users that design, build and experiment with high-performance radios. The openHPSDR domain hosts an active e-mail reflector where new hardware is proposed, software is discussed and where users can get (and offer) openHPSDR system help and operating tips.

System Architecture

From the beginning, the openHPSDR project was designed to be modular and expandable. This type of architecture makes the system a bit more costly and complex because common interface circuitry must be duplicated on each module. However, the resulting system is inherently upgradeable and flexible; these two features are highly desirable from an experimenter's point of view. From the openHPSDR perspective, performance generally takes precedence over cost.

An example of the value of the openHPSDR upgrade path is in order. A production run of the Penelope transmitter board was made by TAPR in May 2008. Penelope was a good transmitter, but it had two shortcomings. First, the power output fell off rather quickly above 30MHz due the design of the PA output stage. Second, there was no hardware power-output control. Power output was reduced by scaling the data values sent to the DAC, resulting in increasing quantization errors (and thus, more distortion in the output waveform) as the output power was decreased. In August of 2011, both of these shortcomings were addressed with the production of the Pennylane transmitter board. Pennylane simply replaces Penelope, uses the same firmware and software, but performs better. Interestingly enough, due to the open source nature of this project, Pennylane was produced by iQuadLabs <http://iQuadLabs.com/> and not by TAPR. More on this later. The example here is that drop-in hardware enhancements are possible with a modular architecture that would not be possible with a single-board SDR.

One other hardware feature is worth noting: all openHPSDR boards that plug into the Atlas backplane are a standard size (100mm by 120mm) and use a standard connector (96-pin DIN41612). This makes a common enclosure for all systems feasible.

I have teased you with mysterious talk of Atlas, Penelope and Pennylane long enough. Let's move on to some hardware details. Please follow the link for each hardware component for more detail, schematics, parts lists, and layouts, as appropriate. I will also indicate a source for purchasing bare PCBs, assembled and tested boards or kits, depending on what is available. Most of the hardware described below is released under TAPR OHL. A few designs are under TAPR NCL, but will be moved to OHL when possible.

It is helpful to refer to Figure 1 while reading the board descriptions below to see how each board fits into the complete openHPSDR system.

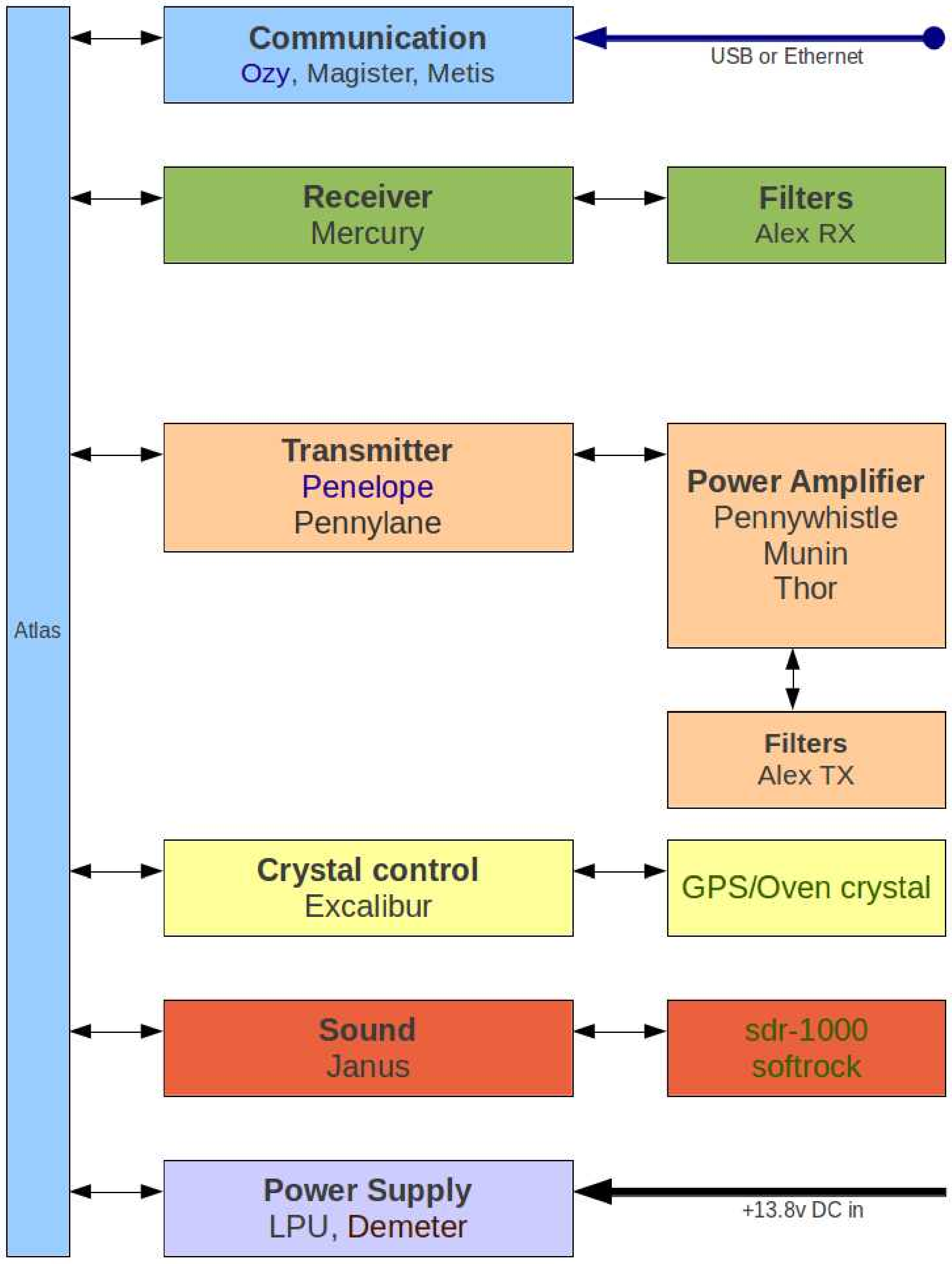


Figure 1. OpenHPSDR System Overview

Hardware – Main Atlas Bus Components

These components consist of the Atlas 6-slot backplane and the three basic boards required for a functioning transmitter/receiver. (It is not a transceiver in the classic sense, since the transmitter and receiver are separate and can operate at the same time, i.e., in full-duplex). All openHPSDR systems must have a communications interface. A receiver, a transmitter or both is also required.

Atlas Backplane – the heart of it all

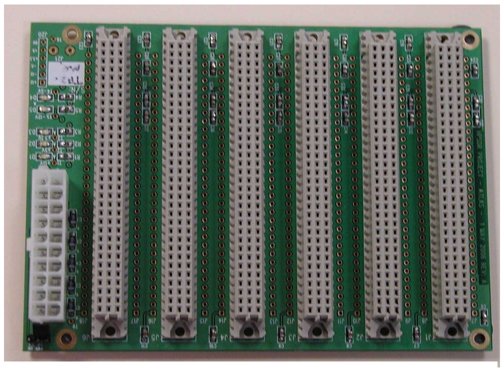


Photo 3. Atlas 6-Slot Backplane

The Atlas 6-slot backplane (Photo 3) consists of six 96-pin DIN connectors bused to an unterminated 48-bit (6-byte) wide bus. In addition to the bus, six pins of each connector are daisy-chained to the adjacent DIN connectors (3 pins to each side). Power connections are provided to each DIN connector for one common ground and five power supplies: +12V, +5V, +3.3V, -5V and -12V. Paralleled pins allow each power supply connection to carry in excess of 2.5A. The power input connector is a standard ATX computer motherboard connector; an off-the-shelf ATX supply can be connected here, but be aware that most PC power supplies are very RF noisy and may compromise receiver small-signal performance. LEDs are provided for each power rail, and a header is provided for remote power control of an ATX supply. Power rail bypassing is abundant.

TAPR offers the Atlas 6-slot backplane as a kit only. The DIN and ATX connectors are through-hole, and the remaining parts are relatively easy to assemble 0805 size SMT parts.

Atlas additional documentation: <http://openhpsdr.org/atlas.php>
Atlas kits: <http://tapr.org/kits_atlas>

Communications Interface – two to choose from

The communications interface is the openHPSDR endpoint of the data path between the PC and the radio. Magister uses USB 2.0 as its interface, while Metis uses Gigabit Ethernet for the same function. Speeds and protocols differ between the two boards, but the function is the same.

Note that these two boards are ***alternates***; you cannot use both interfaces at the same time.

Magister USB Interface

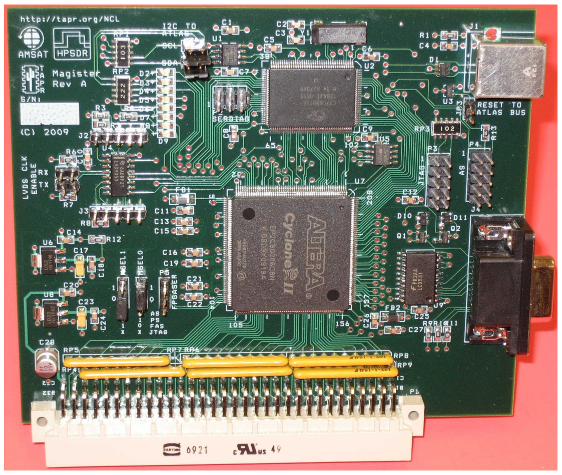


Photo 4. Magister USB Interface

Magister (Photo 4) is a high-speed USB 2.0 interface built around a Cypress FX2 (CY7C68013A) micro-controller and an Altera Cyclone II FPGA (EP2C8). The FX2 provides the USB 2.0 interface to the PC and a FIFO interface to the FPGA. The FPGA formats the data to/from the various openHPSDR components via the Atlas bus.

iQuadLabs offers Magister fully assembled and tested.

Magister additional documentation: <http://openhpsdr.org/magister.php>
Magister boards, assembled and tested: <http://iquadlabs.com/content/magister>

Metis Gigabit Ethernet Interface

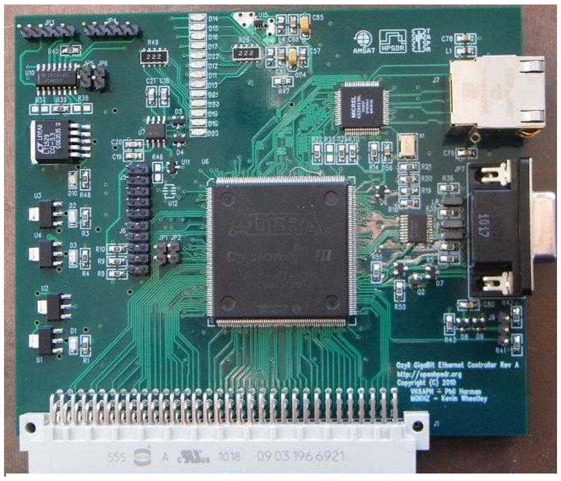


Photo 5. Metis Gigabit Ethernet Interface

Metis (Photo 5) is a 100M/1000M Ethernet interface built around a Micrel KSZ9021RL Gigabit PHY and a large Altera Cyclone III FPGA (EP3C40). The FPGA is the largest Cyclone III part that is available in a leaded (240-pin QFP) package. There are 12 FPGA-controlled LEDs, a LVTTL-level serial port, 512K bytes of SRAM as well as some digital I/O (four outputs and three inputs).

Metis can use an IP address obtained via DHCP; lacking a DHCP server on the network, it will use an APIPA address of the form 169.254.x.x, where x.x is determined by the board’s MAC address. Each Metis board has an on-board EEPROM that is pre-programmed with a unique MAC address. A fixed IP address can be optionally stored in this EEPROM as well. Data from the Atlas bus (from a Mercury receiver, for example) is formatted by the logic in the FPGA and sent to the PC via UDP packets. In the opposite direction, UDP data from the PC is formatted and sent to the Altas bus (to a Pennylane transmitter, for example).

TAPR offers Metis fully assembled and tested.

Metis additional documentation: <http://openhpsdr.org/metis.php>
Metis boards, assembled and tested: <http://tapr.org/kits_metis>

Mercury Direct Sampling Receiver

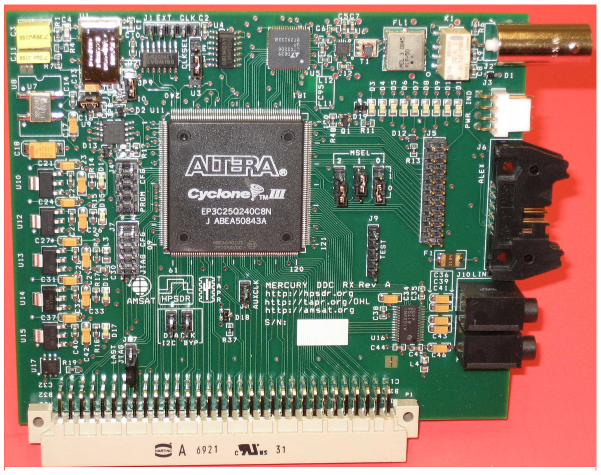


Photo 6. Mercury Direct-Sampling Receiver

Mercury (Photo 6) is a high-speed, direct-sampling receiver board. The Mercury front-end consists of a switchable 20dB attenuator followed by a 20dB LNA (LTC6400-20) and a low-pass filter (LPF). The LPF feeds a 700MHz bandwidth 16-bit ADC (LTC2208) clocked at 122.88MHz. The digitized data from the ADC is fed to an Altera Cyclone III FPGA (EP3C25) where it is processed and sent to the communications interface (Magister or Metis) via the Atlas bus. This "processing" consists of combined filtering and decimation to reduce the amount of data sent across the Atlas backpane and eventually, to the PC for demodulation and/or display. For those interested in the inner workings of the Mercury FPGA code, here is a link:
<http://openhpsdr.org/wiki/index.php?title=Mercury_-_Development_History>

Mercury is a very high performance receiver, with a minimum discernable signal (MDS) of about –138dBm and a blocking dynamic range (BDR) of about 119dB. The BDR is determined by the overload point of the ADC at -12dBm (+8dBm with attenuator switched in) rather than being phase-noise limited. Here is an excellent evaluation of Mercury performance:
<http://openhpsdr.org/wiki/index.php?title=Mercury_-_intermodulation_(IMD)_tests>

iQuadLabs offers Mercury fully assembled and tested.

Mercury additional documentation: <http://openhpsdr.org/mercury.php>
Mercury boards, assembled and tested: <http://iquadlabs.com/content/mercury>

Pennylane Direct Up-Conversion (DUC) 500mW Transmitter

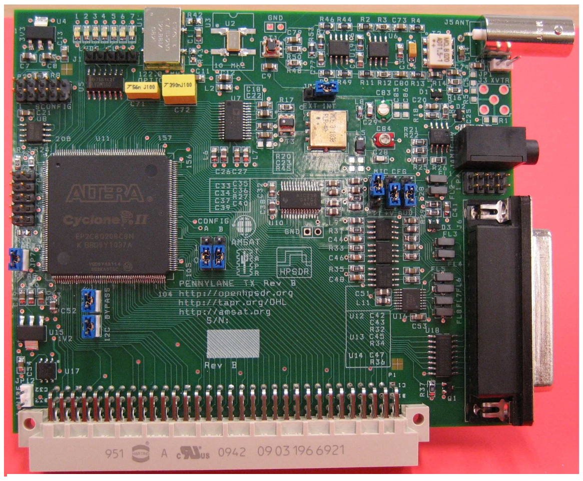


Photo 7. Pennylane 500mW DUC Transmitter

As mentioned above, there are two openHPSDR transmitter boards. The original Penelope transmitter board has been superseded by the new improved Pennylane transmitter. The function of the two boards is identical; Pennylane just does the job a bit better than Penelope.

Pennylane (Photo 7) is a 500mW direct up-conversion transmitter. The transmit data stream from the Atlas bus is processed by an Altera Cyclone II FPGA (EP2C8) and fed to a high-speed 14-bit DAC (AD9744ARU) clocked at 122.88MHz. The analog waveform from the DAC is filtered and then amplified by a two-stage 500mW RF power amplifier (PA). Other features of Pennylane are on-board analog output level detection, four general-purpose analog inputs, three PWM outputs for future class E amplifier support, seven open-collector digital outputs and a CODEC for microphone audio input and auxiliary audio output. The on-board 122.88MHz TCXO is the same ultra-low phase noise type that Mercury uses.

iQuadLabs offers Pennylane fully assembled and tested.

Pennylane additional documentation: <http://openhpsdr.org/penny.php>
Pennylane boards, assembled and tested: <http://iquadlabs.com/content/pennylane>

Hardware – Other Atlas Bus Components

These components consist of a power supply and various other boards that provide additional openHPSDR functions. Excalibur provides enhanced frequency accuracy capability, Janus provides baseband A/D and D/A capability and Pinocchio allows openHPSDR cards to be “extended” above the backplane for debug tasks.

LPU Linear Power Unit

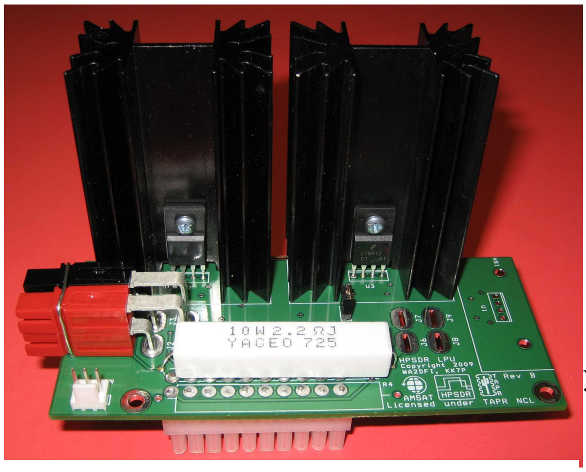


Photo 8. LPU Linear Power Unit

LPU (Photo 8) is a linear regulated power supply designed to power an openHPSDR radio from a regulated 13.8V bench supply. It provides 2A@+12V, 1.5A@+5V and 100mA@-12V from a 12.5V to 14.5V input. LPU can also supply 1A@3.3V if optional parts are installed. The -12V regulator is an inverting switch-mode regulator, and can be disabled to reduce switching noise when -12V is not required. (Janus is the only openHPSDR board that uses -12V.)

LPU passes the regulated input connection through to an internal connector for use by a power amplifier. LPU also provides a header for a 12VDC fan, which is almost always required due to the large amount of heat generated by the linear nature of LPU regulators. LPU plugs directly into the power connector on the Atlas bus, without any cables.

LPU was intended to be a temporary solution until a custom openHPSDR switching power supply could be designed. The switching supply solution has not materialized thus far. LPU is inefficient, but it is also very RF-quiet; this is a good thing for so sensitive a receiver as Mercury.

TAPR offers the LPU power supply as a kit only. The SMT parts are relatively easy to solder 0805 or larger size.

LPU additional documentation: <http://openhpsdr.org/LPU.php>
LPU kits: <http://tapr.org/kits_lpu>

Excalibur 10MHz Frequency Reference

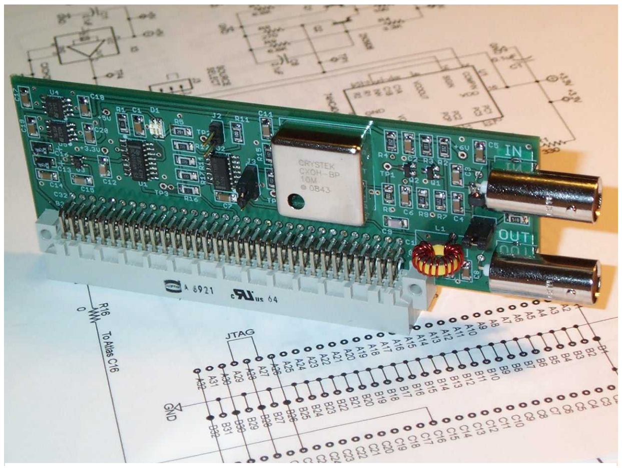


Photo 9. Excalibur Frequency Reference

Excalibur offers two options for generating a precision 10MHz reference clock source for openHPSDR boards. The first option is Excalibur's on-board high-stability 10MHz TCXO, which can be phase-locked to an external input. The second option is an external GPS-disciplined or other precision oscillator. The 10MHz oscillators on Pennylane and Mercury have a rated stability of between +/-50ppm and +/-100ppm. Thus the 10MHz clock error can be up to 1kHz at temperature extremes. Excalibur's TCXO is rated at +/-1ppm, or 10Hz at temperature extremes. At room temperature, the error is typically less than 1Hz.

If an external high-performance GPS disciplined oscillator is used, typical accuracies of +/-0.0001ppm can be reached. This is one milliHertz at 10MHz! The time-nuts <http://leapsecond.com/time-nuts.htm> have lots of fun with Excalibur.

TAPR offers Excalibur as a kit only. The SMT parts are mostly easy to solder 0805 size, but there are a few smaller ICs. There is one evil toroid to wind.

Excalibur additional documentation: <http://openhpsdr.org/excalibur.php>
Excalibur kits: <http://tapr.org/kits_excalibur>

Janus Baseband A/D and D/A Converter

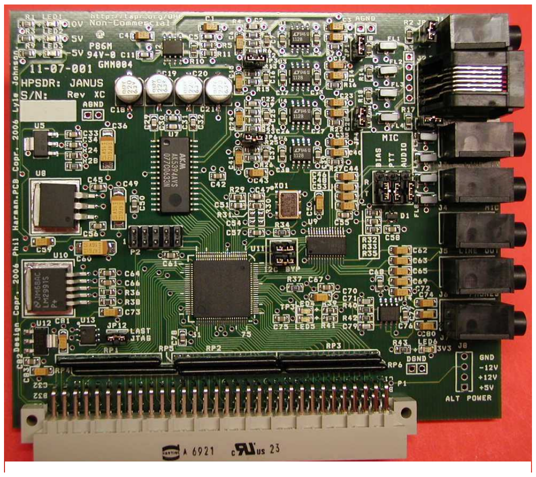


Photo 10. Janus A/D and D/A Converter

Janus (Photo 10) is a very high-performance baseband A/D and D/A (i.e., sound card). It uses a high-performance, 24-bit, 192ksps ADC (AKM AK5394) for baseband input, and a stereo CODEC (TI TLV320AIC23B) for mic/line input and headphones/line output. Janus is intended to be used with a source of I/Q data from a QSD-based receiver. Two examples of such receivers are the Softrock series from Tony Parks, KB9YIG <http://kb9yig.com/> and the SDR-1000 from FlexRadio Systems® <http://www.flex-radio.com/>. Janus' CODEC output can also drive the QSE-based transmitter section of these same radios.

The performance of Janus equals or exceeds all but the very highest performance (read: expensive) PC sound cards. However, there is currently no Windows sound card driver for Janus. It can only be used with software that supports it directly, such as the openHPSDR version of PowerSDR™.

TAPR offers Janus bare PCBs, as well as fully assembled and tested units.

Janus additional documentation: <http://openhpsdr.org/janus.php>
Janus boards, assembled and tested: <http://www.tapr.org/kits_janus>

Pinocchio Extender Card

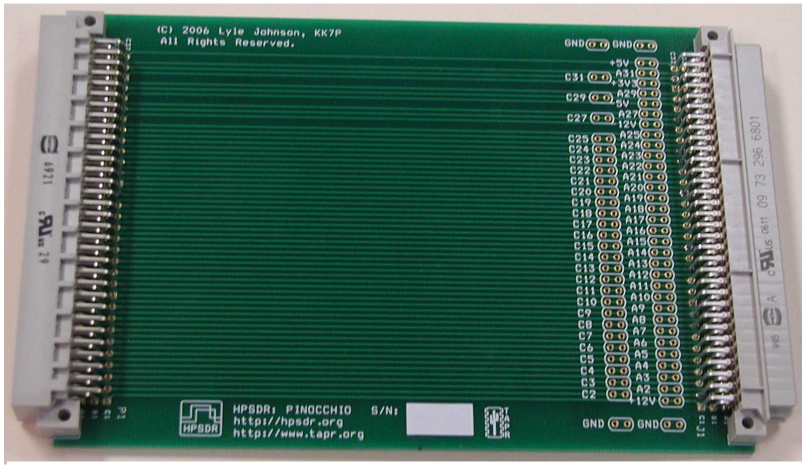


Photo 11. Pinocchio Extender

While Pinocchio was designed to raise any Atlas plug-in card up and into the open so it can be probed, it also has other uses. Since every Atlas bus signal is available on the surface of the card, Pinocchio can make an excellent base for prototyping new hardware. It is a very simple kit, with a through-hole right-angle 96-pin DIN connector on each end of a PCB.

TAPR offers the Pinocchio extender as a kit only.

Pinocchio additional documentation: <http://openhpsdr.org/pinocchio.php>
Pinocchio kits: <http://www.tapr.org/kits_pinocchio>

Hardware – Non-Atlas Bus Components

These components do not plug into the Atlas backplane, but are useful and/or necessary to build up a complete openHPSDR radio.

Pandora Chassis Enclosure

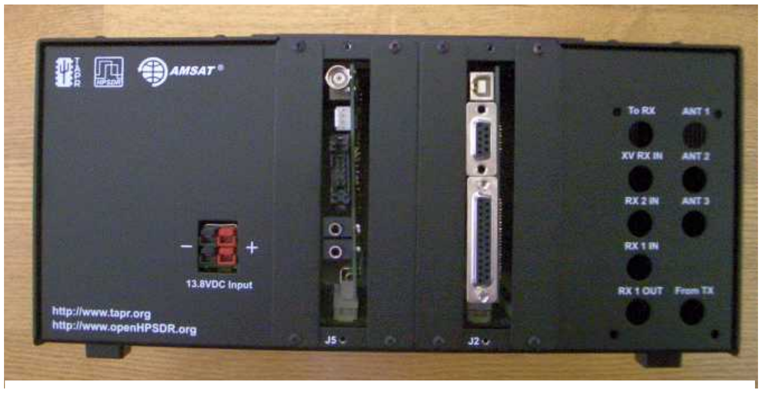


Photo 12. Pandora Enclosure

Pandora (Photo 12) is an enclosure for openHPSDR components. It is large enough to house all of the components necessary for a 20W (or more) HF/6M transceiver. There are provisions for a fan, an Atlas backplane fully loaded with six boards, an LPU, a power amplifier (Pennywhistle or other model) and a set of Alex filters in a shielded sub-chassis. Pandora has a black powder-coated finish and is made of aluminum for easy modification. It is pre-punched and drilled for all of the above components. Blank filler panels are included to block off unused Atlas slots.

TAPR offers Pandora as a bolt-together enclosure complete with hardware and blank filler panels.

Pandora additional documentation: <http://openhpsdr.org/pandora.php>
Pandora kits: <http://www.tapr.org/kits_pandora>

Pennywhistle 20W Power Amplifier

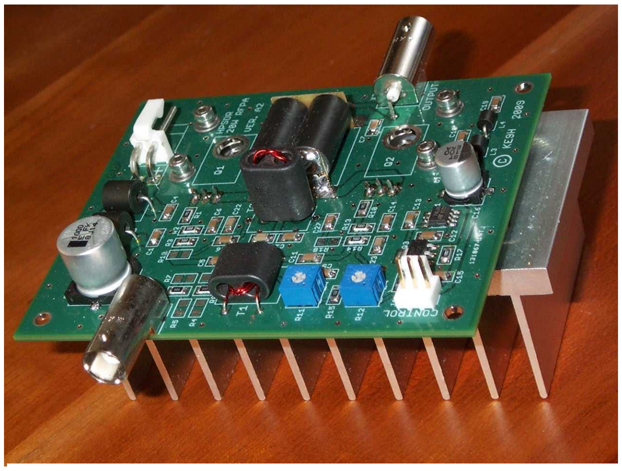


Photo 13. Pennywhistle 20W RF Power Amplifier

Pennywhistle (Photo 13) is an RF power amplifier that produces up to 20W of RF output from 250mW of drive. It uses two RD15HVF1 power MOSFETs in a push-pull output stage and delivers about 19dB of gain. Some kind of low-pass filtering is required (such as Alex, below) to meet FCC regulatory requirements for harmonic emissions.

TAPR offers Pennywhistle as a kit only. The SMT parts are easy to solder 1206 size, and there are a few simple transformers to wind.

Pennywhistle additional documentation: <http://openhpsdr.org/pennywhistle.php>
Pennywhistle kits: <http://www.tapr.org/kits_pw>

Alexiares Transmit/Receive Filters

Alexiares (Alex for short) is a set of filter boards for the openHPSDR project, but these two boards offer much more that just filtering.

The Alex-TX board (Photo 14) not only contains six switched 100W transmit low-pass filters, it has a transmit/receive (T/R) relay, an unswitched 6M LPF, a directional coupler for power measurements and relays to select one of three separate antennas.

The Alex-RX board (Photo 15) contains five switched receive high-pass filters, a 6M LNA, an unswitched 55MHz LPF, a switchable 0/10/20/30 dB attenuator and connections for a transverter, two separate receive antennas and an external filter or preamplifier.

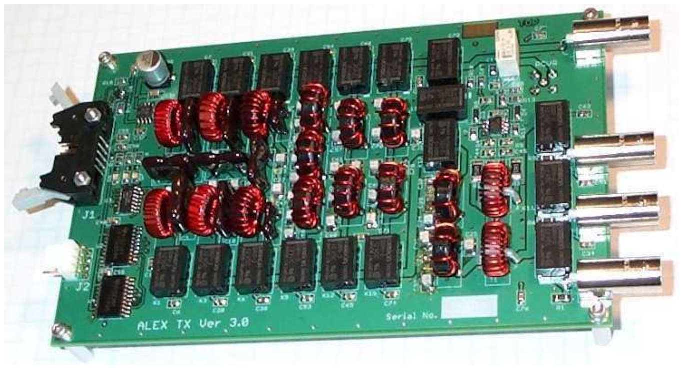


Photo 14. Alex-TX Low-Pass Filter Board

Alex-TX and Alex-RX daisy-chain together on a 10-pin ribbon cable that supplies power and serial control from an interface on the Mercury receiver board. An off-the-shelf extruded aluminum enclosure with custom end plates is available to mount and completely shield the pair of boards in one enclosure.

The Alex-TX and Alex-RX boards are mounted back-to-back in the enclosure. The PCB layers are arranged on each board to shield the transmit components from the receive components. Here are the Alex testing results performed by John Ackerman, N8UR, using laboratory-grade test equipment:
<http://www.febo.com/pages/alex/>

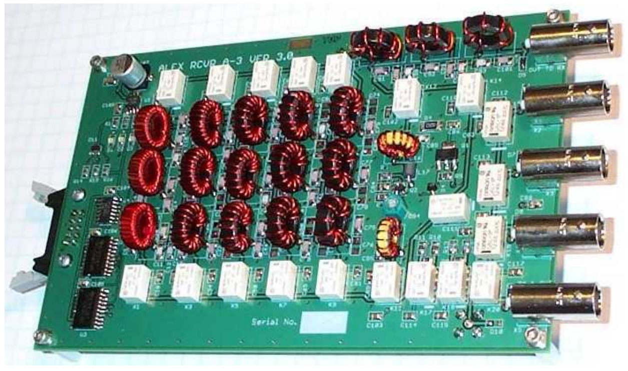


Photo 15. Alex-RX High-Pass Filter Board

TAPR offers Alex-TX and Alex-RX boards fully assembled and tested. TAPR also offers an enclosure with custom end plates for proper shielding. Note that an enclosure is necessary for Alex boards even if they are mounted within the Pandora enclosure for RF shielding reasons.

Alexiares additional documentation: <http://openhpsdr.org/alex.php>
Alexiares TX/RX filter boards/enclosure, assembled and tested: <http://www.tapr.org/kits_alex>

Hardware – Single Board openHPSDR

It is not quite a single board when you include the Apollo 15W power amplifier and automatic Antenna Tuning Unit (ATU), but Hermes does include both the transmitter and receiver on one board. The combination of Hermes and Apollo fits in a standard Eurocard enclosure, yielding a compact and complete 15W high-performance software defined radio. Just how does it compare to the openHPSDR Atlas system? Read on…

Hermes 500mW DUC Transmitter/DS Receiver

Photo 16. Hermes Transmitter/Receiver

The Hermes (Photo 16) receiver section uses the same front-end filter, preamp (LTC6400) and ADC (LTC2208) that Mercury uses. The Hermes transmitter section uses the same DAC (AD9744ARU), filter and RF power amplifier (OPA2674) that Pennylane uses, as well as the same audio CODEC (TLV320AIC23B) and analog input circuit (ADC78H90). The Hermes Ethernet interface uses the same PHY (KSZ9021RL) that Metis uses. The three FPGAs from Metis, Mercury and Pennylane (EP3C40, EP3C25 and EP2C8, respectively) are replaced by a single EP3C40 FPGA. The new layout is really the only variable, and preliminary testing indicates that Hermes is actually quieter on receive than Mercury and has transmit performance equivalent to Pennylane.

Hermes additional documentation: <http://openhpsdr.org/hermes.php>
Hermes boards, assembled and tested: **(under development, TAPR will be the likely source)**

Apollo 15W Power Amplifier/LP Filter Bank/Automatic Antenna Tuner

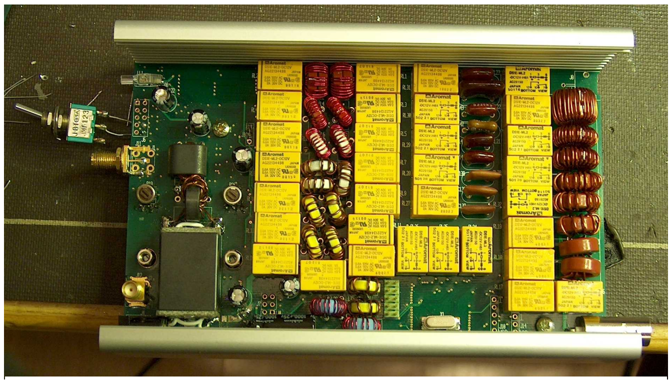


Photo 17. Prototype Apollo 15W PA/ATU

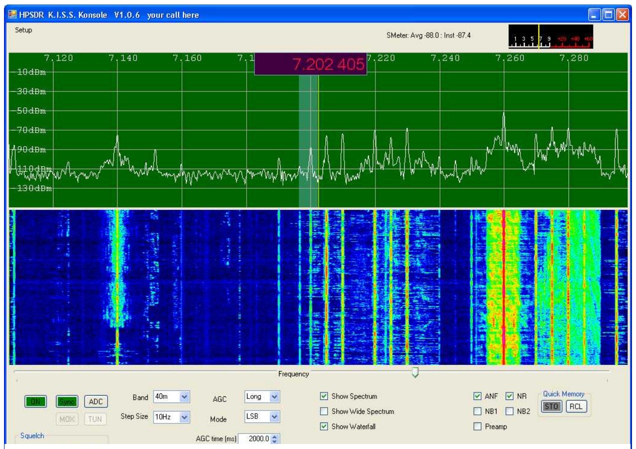
Apollo (Photo 17) is a companion board to Hermes, and boosts Hermes' 500mW RF output to 15W with a pair of RD15HVF1 MOSFETs in a push-pull amplifier configuration. Apollo contains a set of low-pass filters to reduce transmitter harmonic energy. These LP filters are low-power versions of the filters on the 100W Alex-TX board. Apollo has an automatic antenna tuner (ATU) that uses an Atmel AT90 micro-controller in conjunction with an on-board directional coupler to determine the output mismatch and then switch in capacitance and inductance to correct it. Switching is done with latching relays to conserve power. The result is a power amplifier correctly matched and harmonically filtered.

Apollo additional documentation: <http://openhpsdr.org/wiki/index.php?title=APOLLO>
Apollo boards, assembled and tested: **(under development, TAPR will be the likely source)**

openHPSDR Software

The focus of this article has obviously been on the hardware, but since openHPSDR is a ***Software*** Defined Radio, it stands to reason that there must be *some* software involved. Several good programs are available that allow most everyone to play, whatever your computing persuasion.

For Windows PC users, you can use a derivative of PowerSDR™, the GPL software that was developed by FlexRadio Systems® for their product line. This software was originally modified by Bill Tracey, KD5TFD, to support openHPSDR software. The software is currently supported by Doug Wigley, W5WC and is at revision 2.23. It is full featured and works very well in the Windows XP and Windows 7 environments. Joe Martin, K5SO, has a modified version of PowerSDR™ 2.2.3 that works with multiple Mercury boards for diversity reception and beam steering. More information on all of these variants of PowerSDR™ for openHPSDR can be found here: <http://openhpsdr.org/wiki/index.php?title=PowerSDR>



SMeter: Avg -88.0 : Inst -87.4

Band 40m
Step Size 10Hz
AGC Long
Mode LSB
AGC time (ms) 2000.0
☑ Show Spectrum
☐ Show Wide Spectrum
☑ Show Waterfall
☑ ANF ☑ NR
☐ NB1 ☐ NB2
☐ Preamp

Photo 18. An Early Version of KISS Konsole

Kiss Konsole (KK for short) is a basic program for beginners to get their feet wet in SDR and DSP programming written by Phil Harman, VK6APH. It is written for Windows in C# using the free VS 2008 IDE. It is heavily commented and is a good starting point for new SDR programmers. Further information on KISS Konsole can be found here: <http://openhpsdr.org/wiki/index.php?title=KISS_Konsole>

For Linux users, John Melton, GØORX/N6LYT has written two versions of openHPSDR software: GHPSDR standalone and GHPSDR3 server/client. The stand-alone GHPSDR was developed on the Ubuntu version of Linux (specifically version 9.04). This code runs on MacOS as well. You can find more information on GHPSDR here: <http://openhpsdr.org/wiki/index.php?title=Ghpsdr>

GHPSDR3 is a client/server implementation that allows the server and client to be either on the same machine or on separate machines. The servers are written in C and run on Linux machines (specifically Ubuntu version 9.10). They have also been ported to Windows. There are several variants of clients, either completed or under development, notably a Java version (jmonitor) and a Qt4 version (qtmonitor). Qt4 is an open-source cross platform environment, so the code compiles and runs on Linux, Windows and MacOS. GHPSDR3 information can be found here: <http://openhpsdr.org/wiki/index.php?title=Ghpsdr3>

Dave McQuate, WA8YWQ, has developed a variant of GHPSDR3 server called ghpsdr3-Windows that will run up to four virtual receivers within one Mercury board. This code is

complied for Windows and can be found in the SVN repository here:
<svn://64.245.179.219/svn/repos_hpsdr_kiss/branches/WA8YWQ/ghpsdr3-Windows>

MAC users have another option besides GHPSDR. Jeremy McDermond, NH6Z, has written a version of openHPSDR software just for you: Heterodyne. Formerly called MACHPSDR, this software runs on Snow Leopard (MacOS X 10.6) for Intel systems. Here is a link to more information: <http://openhpsdr.org/wiki/index.php?title=MacHPSDR>

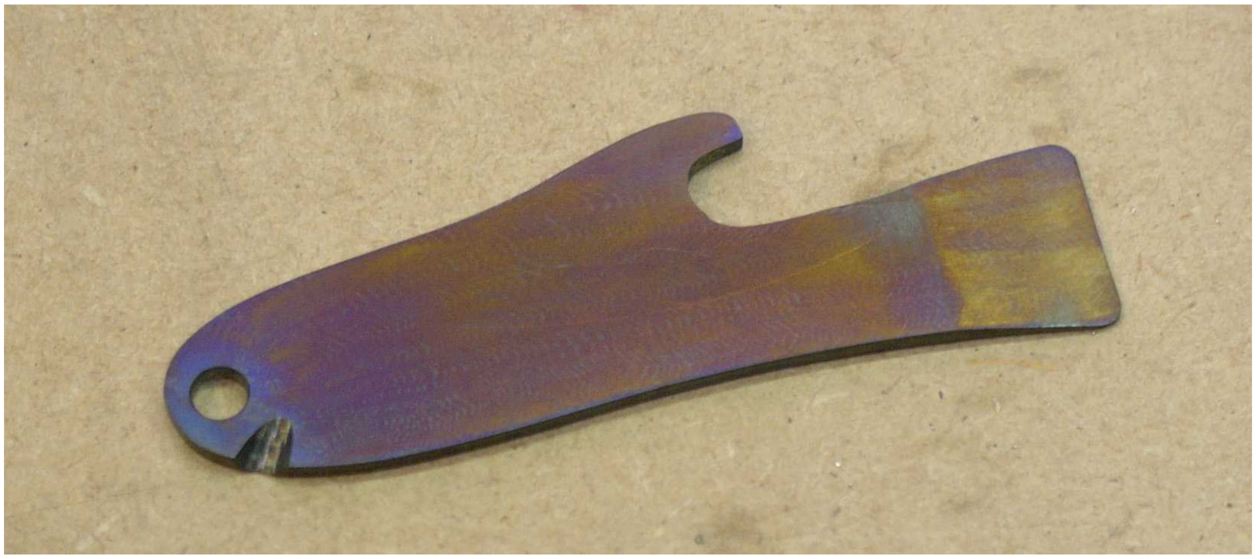
Conclusion

The openHPSDR project is an ongoing evolution of ideas and implementation. The best place to jump in is the OpenHPSDR e-mail list. Please come join us! You can subscribe here:
<http://openhpsdr.org/reflector.php>

Useful links

openHPSDR web site: <http://openhpsdr.org/>
openHPSDR Wiki: <http://openhpsdr.org/wiki/index.php?title=HPSDRwiki:Community_Portal>
openHPSDR hardware from iQuadLabs: http://iquadlabs.com/
openHPSDR hardware from TAPR: <http://tapr.org/>
TAPR open hardware license: <http://tapr.org/ohl>

metalfishy: A Pocket Tool for Everyday Carry


Title Illustration: A Titanium metalfishy

By Tait Stevens (tait.stevens@gmail.com) and Loren Cress (lpcress@yahoo.com)

History

Life currently seems to involve opening lots of boxes and packages. We have carried small pocket knives; however, it's not fun forgetting to remove it before airport security and having to choose between keeping a $20 pocket knife and catching a $400 flight. Car keys work pretty well for cutting tape and opening boxes, but they can be made from soft metal that wears easily – to the point that they no longer work to unlock the car.

We decided to explore alternatives that would fit on a keychain for convenient every day carry.

There are some commercial options, including offerings from companies such as Pocket Tool X, Gerber, and Swiss Tech, and individual manufacturers, such as Peter Atwood. None of these appear to be open source products.

First version

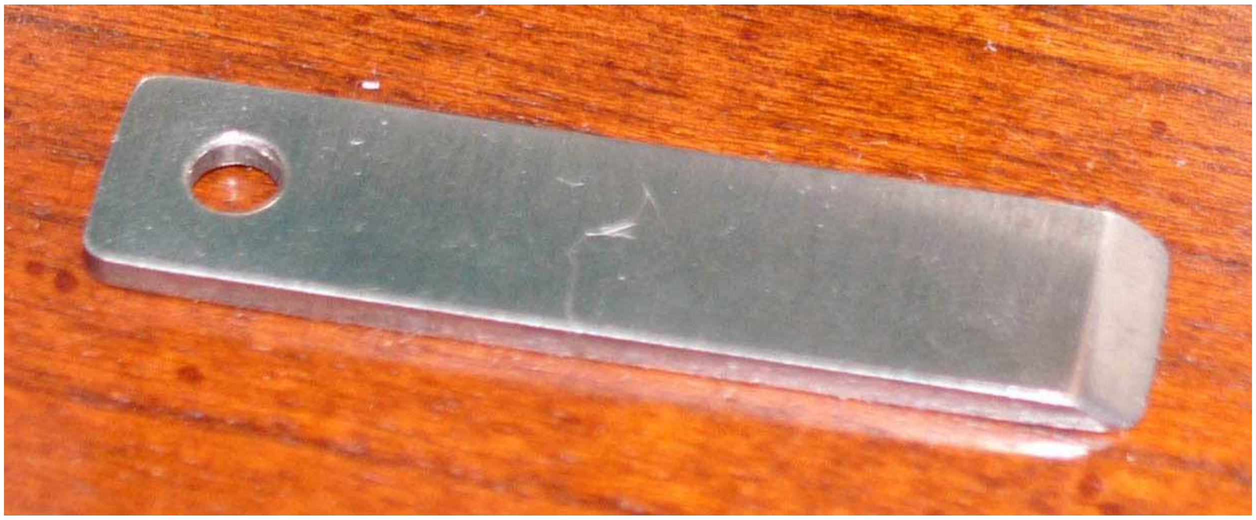


Illustration 1: A "chiselette"

We first a miniature chisel blade – a “chiselette” (Illustration 1). Hardening tool steel was chosen as it would be fairly wear resistant. A 2.5” piece was cut from 3/32” x 1/2” bar stick. A hole was drilled in one end, the other sharpened using a belt sander, and then the steel hardened per manufacturer instructions. After light sanding to round sharp edges, the resulting miniature chisel fits nicely on a keychain. We ended up making several dozen of these for friends and family, who found them useful for many activities other than just opening boxes.

Current version

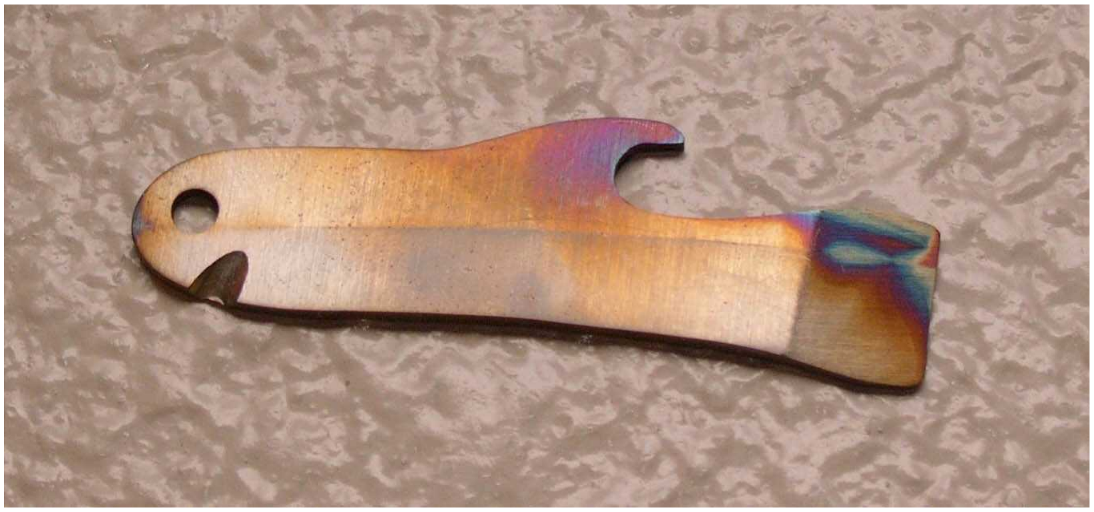


Illustration 2: a “metalfishy”

Ongoing reflection on the tool lead to recognition that a bottle opener could be useful if incorporated into the design. Multiple prototypes lead to the current fish shape with a fin as the bottle opener (Illustration 2). This design has been dubbed “metalfishy” by a 2 year old family member. We distribute this design with a card naming the uses of various features (Illustration 3). These have been even more popular with friends, some of whom have suggested marketing the fish-shaped bottle opener commercially. Several have been carried through airport security multiple times without incident.

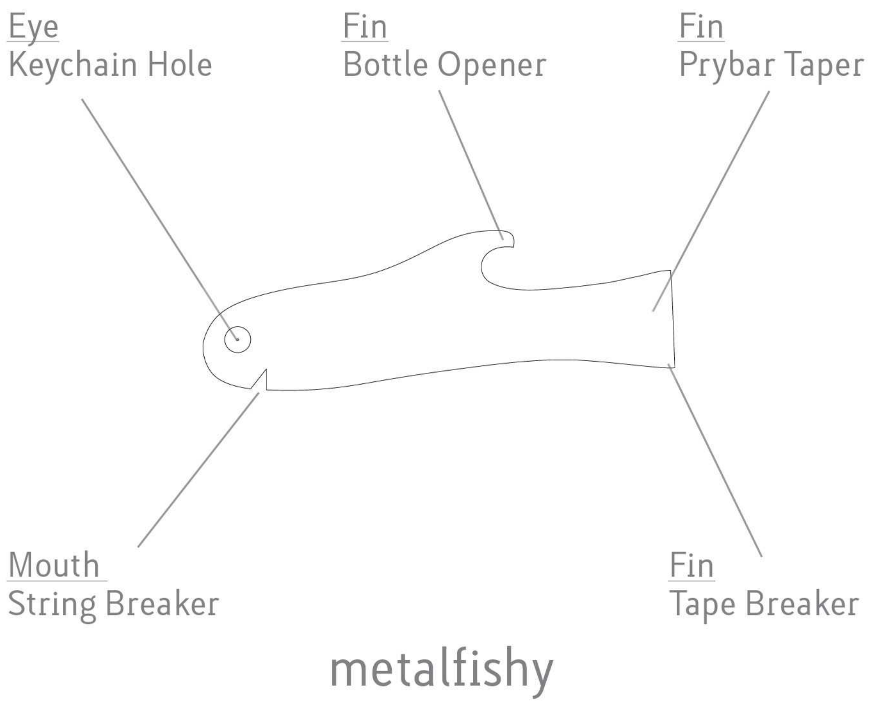


Illustration 3: metalfishy features

Uses

The metalfishy's primary uses are as a bottle opener and tape breaker/box opener. Other reported uses: the bottle opener can be used to lift sealed lids on canned goods, the chisel fin works very well for scraping small areas (such as vagrant paint flecks), as a metal fingernail (such as to remove batteries that need changing), and as an impromptu screwdriver (in one case, to fix a broken hotel showerhead).

Material selection

The metalfishy's shape can be made from essentially any flat metal stock. Harder metals, such as steel and titanium seem likely to last longer than softer selections. The most wear resistant have been made from high alloy steel requiring heat treatment in excess of 1,500F to harden fully. We have made one from titanium (title illustration). Titanium stock is readily available for order on the internet and seems to be holding up very well.

Making metalfishies

Minimal equipment for making one to two would likely include a drill press, hacksaw or bandsaw, and files. Additional observations are available as part of a documented work in progress showing some images from the making of a titanium metalfishy at http://taitstevens.com/metalfishy.

The design relatively simple and there is enough surface area that additional decoration is feasible, such as electric coloration of titanium (title illustration) or etching of figures (such as inch or centimeter scales[1]).

Contact

Making and using a hand-made, open-source pocket tool is a lot of fun. We would appreciate a note from anyone who makes and uses metalfishies, particularly if the design is improved upon. We will also be happy to answer any questions.

1 Vile pun.

The Insufficiency of the AppNote and its replacement by Open Source Hardware: as Shown Through a LIN Protocol Implementation

A significant cost to "hardware development" is in fact the software that runs on it. All major embedded microcontroller manufacturers spend significant resources on what is generally termed "Application Notes" or AppNotes that often feature their hardware running software required for some specific market. These AppNotes essentially act as proof-of-concept work to show that the hardware is capable of performing the needed functions. Generally, these implementations are license-locked to a particular manufacturer's microcontroller and are often not production ready. After all, the perceived value to the manufacturer is primarily that of an advertisement – and as an advertisement the added value in fixing stability and other issues is practically zero. Also, for a variety of reasons (budget, engineer quality, and simply thousands of devices to expose rare bugs) it is very difficult to create a production-quality implementation without actually going into production!

Meanwhile, the AppNote becomes much less valuable once many other manufacturers provide competing AppNotes. It does not become worthless – instead it becomes a barrier to entry since late-comers must offer competing AppNotes so their microcontroller will be evaluated for the chosen application.

This means that the industry as a whole ends up with multiple implementations of the same (or similar) functionality, none of which are actually usable by the engineers who need it due to issues with stability, quality, and licensing. These engineers therefore create their own robust implementations. But these robust implementations are never shared; it is clearly directly against the economic interest of a traditional end-device manufacturer to share the software which can be used to produce a competing device.

Game theorists call this situation a state of "Nash equilibrium". In summary, Nash equilibrium occurs in a situation where all players have made the best possible decision based on their knowledge of other player's decisions, but in fact the overall situation may not be optimal for any of the players. In other words, the advantage to a manufacturer of being first to market for a particular AppNote topic is overshadowed both by not being first in other AppNote topics and by providing a poor implementation. A poor implementation ultimately increases end-product development time and quality and therefore can affect the time before volume chip purchases begin. The end-device manufacturer has higher front-end costs, resulting in a longer time-to-market and higher shelf price.

The industry as a whole could save a lot of time and effort by producing a single, robust, implementation of whatever topic is covered by the AppNote, with a low level "compatibility layer" that glues this common code to the specific registers of each microprocessor. However, that would require one of the players to act against their immediate economic benefit by essentially giving away an AppNote implementation! Open Source Hardware is the concept that breaks this stalemate.

An Example: The LIN Protocol

A Glance At LIN

The LIN protocol was originally made for automotive use but in fact it is pretty useful for any distributed wired sensor or actuator network, especially within a space-constrained electrically noisy environment such as a robot or automobile.

It is a clever transformation of the ubiquitous UART hardware into a single wire packetized, master/slave protocol for control of many devices distributed throughout your project. The protocol essentially fills the same niche as I2C does in a PCB, but for longer wire runs. The most common solution (used here) requires 3 wires: power, LIN, and ground, but chips exist to multiplex the LIN signal onto the Vcc wire, resulting in a 2 wire total solution. In fact, the protocol is carefully constructed to not require a UART or even a reliable (crystal) clock on the slave nodes, allowing for extremely cheap slave devices. But this causes a design tradeoff -- it is not intended for significant data transfer, as it is limited to a theoretical bit rate of 19.2kbaud. And in practice a lot of those bits are protocol overhead, not application data.

On the hardware side an inexpensive, generally 8-pin chip converts the microcontroller's TX and RX lines (or any general purpose IO) into a signal on the LIN bus wire. I used the Atmel ATA6663 chip, but similar 8-pin chips are available from many manufacturers.

The LIN Frame

Software in the microcontroller transmits a special “packetized” protocol over the UART that looks like:

A BREAK signal: Pulls the TX line low (which is actually the dominant, or logical “1” signal) for longer then is allowed in UART to signal start-of-frame. This is the only non-UART part of the signal. All subsequent bytes are transmitted as payloads of the UART protocol. To aid in clarity, these UART envelope bits will not be mentioned in the following description.

A sync byte (0x55): This alternating 1 and 0 byte allows slaves to calibrate their clocks.

An ID byte: Indicates which slave device the master it talking to, and contains some parity bits.

1 to 8 bytes of data: This length is limited by how fast non-crystal clocks typically wander.

A checksum byte: for data integrity.

Higher Layers

The LIN protocol definition also contains a concept of a "schedule table" which is not really part of the protocol; it is essentially a nice design pattern that can be used on the master to achieve determinism when periodically talking to multiple devices. And if you would like to transmit more then 8 bytes, there is a higher layer protocol that can be used to segment and reassemble bigger packets. Of course, this is again an optional protocol in the sense that a master and a slave can use ANY higher layer protocol (if needed) without causing other slave devices to see errors.

So the core LIN functionality is the generation of the LIN Frame.

Existing Application Notes

The LIN protocol contains AppNote implementations by most major embedded mCU manufacturers; a quick search turned up articles and implementations by Atmel[2], Microchip[3], Freescale[4], and Renesas[5]. Many vendors have multiple implementations for different processor families. A quick review shows that these implementations consist of C and assembly code totaling between 1.5k and 5k lines of code per AppNote. This results in a per project cost between 50 and 150 thousand dollars (the differences are due to the different number of lines of code), if you believe the popular line-counting tool "sloccount"[6]. The purpose of these AppNotes is to essentially prove that the microprocessor can operate quickly enough to handle LIN frames, as well as to showcase the particular vendor's LIN transceiver (if it has one).

Typically[7], these projects are licensed-locked onto their manufacturers devices, and contain significant sections in assembly language. A performance-based justification for the use of significant amounts of assembly language is becoming increasingly hard to swallow as embedded CPU speeds are becoming quite fast. But each mCU family uses a different assembly language so the use of assembly language certainly creates strong vendor lock-in. At the same time, it makes it much harder for any engineer who is not an expert in the mCU family (and in assembly language in general) to debug or extend the AppNote code.

Additionally, certain licenses also claim copyright ownership of all derived works (i.e. your work) and require notification to be given to the manufacturer of all such works.

Generally these factors (and the previously mentioned code quality issues) discourage the adoption of AppNote software for an actual product, meaning that every application company must create its own, robust, implementation.

An Open Source Implementation

2 AVR322
3 AN1099, AN237, AN239, AN240, AN729, AN864, AN891
4 68HC908AZ60LINDRV.zip
5 LIN_R8C23_Demo_master_2slave_v1.19.zip
6 http://www.dwheeler.com/sloccount/
7 I checked most but not all projects

LIN Protocol Frames

The basic LIN frame can be constructed in about 100 lines of code when built upon existing Arduino Library APIs. The “Arduino” is an open hardware board, based on the ATMEGA family consisting of open source hardware (schematics and PCB), open source firmware, and open source IDE and toolchain[8]. It is therefore truly an open device. While the “open” aspects of the Arduino hardware platform have been widely praised, the idea of abstracting the hardware into a standard software API “underface” within embedded microprocessors is just as powerful. Embedded microprocessors have traditionally resisted the API standardization that has occurred in smart phones and desktop machines due to size and efficiency concerns. However, today even $3 embedded microprocessors can hold significant amounts of code and RAM. So the benefits of API standardization, like application portability and modular design, outweigh the detriments.

In particular, this LIN implementation will use the Arduino's Serial APIs[9], digital IO APIs, and delay functions. Since the implementation is so short it will be included here, however, please check https://github.com/gandrewstone/LIN for the latest implementation. First, two boring helper functions to generate parity bits and checksums:

```
/* Lin defines its checksum as an inverted 8 bit sum with carry */
uint8_t Lin::dataChecksum(const uint8_t* message, char nBytes,uint16_t sum)
{
    while (nBytes-- > 0) sum += *(message++);
    // Add the carry
    while(sum>>8)  // In case adding the carry causes another carry
      sum = (sum&255)+(sum>>8);
    return (~sum);
}

/* Create the Lin ID parity */
#define BIT(data,shift) ((addr&(1<<shift))>>shift)
uint8_t Lin::addrParity(uint8_t addr)
{
  uint8_t p0 = BIT(addr,0) ^ BIT(addr,1) ^ BIT(addr,2) ^ BIT(addr,4);
  uint8_t p1 = ~(BIT(addr,1) ^ BIT(addr,3) ^ BIT(addr,4) ^ BIT(addr,5));
  return (p0 | (p1<<1))<<6;
}
```

Next, generation of the BREAK signal. This implementation generally follows recommendations posted to the Arduino forum[10] in a conversation about the DMX protocol. The fact that some aspects of this problem was already solved within a completely different application context illustrates another powerful feature of open source firmware – the companion open forums yield synergistic efficiencies.

8 Http://www.arduino.cc
9 In the “standard” Arduino, the serial port is already used to communicate with the developer's PC. For convenience, a clone (the Lightuino, www.toastedcircuits.com) was actually used since it talks to the PC through a USB SPI chip thus freeing up the serial port. Also I have a lot of them :-).
10 http://www.arduino.cc/cgi-bin/yabb2/YaBB.pl?num=1237491111

```
// Generate a BREAK signal (a low signal for longer than a byte) across the
serial line
void Lin::serialBreak(void)
{
  if (serialOn) serial.end();

  pinMode(txPin, OUTPUT);
  digitalWrite(txPin, LOW);  // Send BREAK
  _delay_us((1000000UL/((unsigned long int)serialSpd))*LIN_BREAK_DURATION);
  digitalWrite(txPin, HIGH);  // BREAK delimiter
  _delay_us(1000000UL/((unsigned long int)serialSpd));
  serial.begin(serialSpd);
  serialOn = 1;
}
```

Finally, generation of a LIN “send” frame:

```
/* Send a message across the Lin bus */
void Lin::send(uint8_t addr, const uint8_t* message, uint8_t nBytes,uint8_t
proto)
{
  uint8_t addrbyte = (addr&0x3f) | addrParity(addr);
  uint8_t cksum = dataChecksum(message,nBytes,(proto==1) ? 0:addrbyte);
  serialBreak();       // Generate the low signal that exceeds 1 char.
  serial.write(0x55);  // Sync byte
  serial.write(addrbyte);  // ID byte
  serial.write(message,nBytes);  // data bytes
  serial.write(cksum);  // checksum
}
```

This code is straightforward except for a small detail in the LIN protocol checksum. It changed from version 1 to version 2 to cover the address byte. So the protocol version is passed into the send function and if its version 2, the checksum function is “seeded” with the LIN address byte.

Next, generation of a LIN “recv” frame. This code is more complex for 2 reasons. First, the UART hardware continues to drive the TX pin which overrides any driving by the slaves. So the TX pin must be put into a high impedance (i.e. input) state to release the bus for the slave to drive, and turn it back to an “output” pin when the slave is done. This is implemented with the pinMode() function call.
Second, since LIN is a single wire the UART's RX pin will “hear” what is transmitted on the TX. So some logic is added to discard these echos.

```
uint8_t Lin::recv(uint8_t addr, uint8_t* message, uint8_t nBytes,uint8_t
proto)
{
  uint8_t bytesRcvd=0;
  unsigned int timeoutCount=0;
  serialBreak();       // Generate the low signal that exceeds 1 char.
  serial.flush();
  serial.write(0x55);  // Sync byte
  uint8_t idByte = (addr&0x3f) | addrParity(addr);
  serial.write(idByte);  // ID byte
  pinMode(txPin, INPUT);
  digitalWrite(txPin, LOW);  // don't pull up
  do { // I hear myself
    while(!serial.available()) { _delay_us(100); timeoutCount+= 100; if
(timeoutCount>=timeout) goto done; }
  } while(serial.read() != 0x55);
  do {
    while(!serial.available()) { _delay_us(100); timeoutCount+= 100; if
(timeoutCount>=timeout) goto done; }
  } while(serial.read() != idByte);

  for (uint8_t i=0;i<nBytes;i++)
    {
      // This while loop strategy does not take into account the added time
for the logic.  So the actual timeout will be slightly longer then written
here.
      while(!serial.available()) { _delay_us(100); timeoutCount+= 100; if
(timeoutCount>=timeout) goto done; }
      message[i] = serial.read();
      bytesRcvd++;
    }
  while(!serial.available()) { _delay_us(100); timeoutCount+= 100; if
(timeoutCount>=timeout) goto done; }
  if (serial.available())
    {
    uint8_t cksum = serial.read();
    bytesRcvd++;
    if (proto==1) idByte = 0;  // Don't cksum the ID byte in LIN 1.x
    if (dataChecksum(message,nBytes,idByte) == cksum) bytesRcvd = 0xff;
    }

done:
  pinMode(txPin, OUTPUT);

  return bytesRcvd;
}
```

Higher Layers

The provided implementation also contains a LIN “schedule table”. In summary, the LIN schedule table provides a mechanism to organize a set of messages that need to be repeated periodically so that they do not interfere with each other.   Typically this is used when polling LIN based sensors.

This schedule table is based on a standard skew heap[11] data structure (previously implemented as open source). A skew heap is essentially a self-sorting tree; in this case the root node will always be the next LIN frame that needs to be transmitted. The implementation required about 50 lines of code.

11 http://en.wikipedia.org/wiki/Skew_heap

A Call For Participation

This implementation of the LIN protocol is done in less then 200 lines. This is a factor of 5 to 25 times smaller then other "AppNote" based implementations. Although it cannot be known how much the LIN-related AppNotes have cost chip manufacturer and device makers, it is very likely that the cost to develop this implementation is similarly much smaller.

Manufacturers, this means that it is no longer cost-effective for a chip manufacturer to base an implementation on exclusive code. Instead it makes better economic sense for chip manufacturers to follow these steps:

1. Produce an implementation of the core "Arduino" (or other open source) APIs for the chip. A significant number of engineers are familiar with these APIs, instantly and massively broadening the possible user base for the chip.
2. Test existing "applications" written over these open source APIs on the chip, fix bugs, and release AppNotes describing the work.
3. Fund development of additional AppNotes implemented over these open source APIs. While this can be done in-house, it is more likely that dedicated open source engineers will do a better job. Your core expertise is in hardware design and manufacturing, not software. Development will be much less expensive and will be reusable on your other chips.

Device makers, it makes sense to insist that chip manufacturer follow the process outlined above. Do not let a manufacturer lock you into their architecture with a "gift" of a large and unmaintainable library of inscrutable assembly (or register-banging C code)! Instead, insist on a fully open (not licensed-locked), modular, community-maintained (but possibly manufacturer funded) implementation running on top of open source APIs. This will produce the highest quality software for the lowest cost.

Call for Papers

Deadlines

April 1, 2012 for the May issue.

July 1, 2012 for the August issue.

Email finished papers or correspondence to bruce@perens.com

Papers must be on the topic of Open Hardware. The licensing of the Open Hardware must be compliant with the Open Hardware Definition 1.1 (as it existed on August 2011), at http://freedomdefined.org/OSHW_draft . Design files must be available, please provide the links in your article.

We'd be delighted if you'd use LibreOffice and its OpenDocument file formats to write your article, as that's what we're using. You can also use LibreOffice to convert your article from *Microsoft Word,* etc. If it's too much trouble to use LibreOffice, please use whatever you wish and we'll convert the file. We strongly prefer an editable file format to PDFs.

Articles will remain your property or that of your institution, as you decide among yourselves, and you will retain all rights of a copyright holder, including the right to reprint as you like with no fee to us.

If you don't understand how to license your article, we'll do it for you. All articles must be licensed as specified below.

- The Creative Commons Attribution 3.0 United States (CC BY 3.0) at http://creativecommons.org/licenses/by/3.0/us/

How to Copy This Journal

Credits

Our logo is by Laura Rodríguez, know also as "LiR", of Papermint Designs.

Colophon

This issue was edited and produced on a *Debian* GNU/Linux system. All of the tools used were Open Source / Free Software. Most of them came directly from the Debian distribution. The paper authors also appear to have used Open Source / Free Software tools.

The main editing tool was LibreOffice.