SYSTEM SOLUTIONS FOR DRONES
JANUARY 2020
ISSUE 354

circuitcellar.com

circuit cellar

Inspiring the Evolution of Embedded Design

SYSTEM SOLUTIONS SPEED DRONE DESIGNS

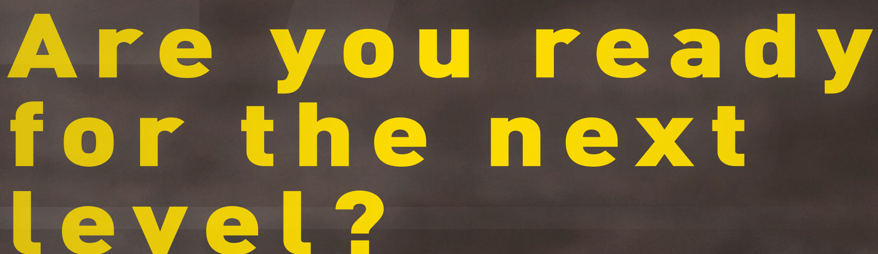
Datasheet: COM Express Boards | Battery Management ICs | Sound Localization | Tips for Choosing Embedded Products | Gesture-Controlled Speakers | System Controller Manufacturing Test (Part 1) | Measuring Air Quality | Design Against FI Attacks | Semi Basics (Part 5) | Smart LEDs (Part 1) | Relaxation Generator Redesigned | The Future of IoT as Safety Resource

$9.00US $10.00CAN
0 74470 75349 0 01

Make your code even faster, smaller, and smarter
while ensuring robustness and high quality.

IAR Embedded Workbench

for RISC-V



As the only commercial tools vendor, IAR Systems is able to provide stable and future-proof technology as well as global technical support. We are specialists on embedded development and help customers when they need it the most, enabling them to make the products of today and the innovations of tomorrow.

We take RISC-V to the next level.

Sign up now for your free evaluation license!

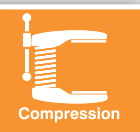
www.iar.com

The Embedded Experts

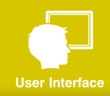
Debug Probes

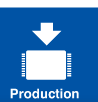
Software Tools

RTOS

File System

Compression

Connectivity

Security & IoT

User Interface

In-Field-Upgrades

Production

emCompress-ToGo

Compress Data in Real-time on any Embedded System!

More Storage

More Bandwidth

Faster Updates

One Professional Compression Solution for All Applications

Data Loggers

Internet of Things

Space / Avionics

Networking

Medical Devices

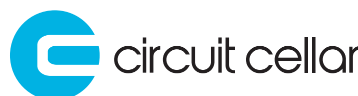
Consumer Electronics

- Real-time compression
- Small footprint
- No static RAM required
- Compression of data stream
- High performance
- High compression ratio
- On-target compression & decompression

Worldwide: sales@segger.com
+49 2173 99312 0
U.S. East Coast: us-east@segger.com
+1 978 874 0299
U.S. West Coast: us-west@segger.com
+1 408 767 4068

segger.com

OUR NETWORK



SUPPORTING COMPANIES

Accutrace, Inc.	C3
All Electronics Corp.	77
CCS, Inc.	77
Earth Computer Technologies, Inc.	23
Embedded World 2020	19
IAR Sytems	C2
Revenue Control Systems	77
SEGGER Microcontroller Systems	1
Siborg Systems, Inc.	31
SlingShot Assembly	67
Technologic Systems, Inc.	C4, 77
University of Cincinnati	11

NOT A SUPPORTING COMPANY YET?

Contact Hugh Heinsohn
(hugh@circuitcellar.com, Phone: 757-525-3677, Fax: 888-980-1303)
to reserve space in the next issue of *Circuit Cellar.*

THE TEAM

PRESIDENT
KC Prescott

CONTROLLER
Chuck Fellows

FOUNDER
Steve Ciarcia

EDITOR-IN-CHIEF
Jeff Child

SENIOR ASSOCIATE EDITOR
Shannon Becker

TECHNICAL COPY EDITOR
Carol Bower

GRAPHICS
Grace Chen
Heather Rennae

ADVERTISING COORDINATOR
Nathaniel Black

ADVERTISING SALES REP.
Hugh Heinsohn

PROJECT EDITORS
Chris Coulston
Ken Davidson
David Tweed

COLUMNISTS
Jeff Bachiochi (From the Bench), Bob Japenga (Embedded in Thin Slices), Robert Lacoste (The Darker Side), Brian Millier (Picking Up Mixed Signals), George Novacek (The Consummate Engineer), and Colin O'Flynn (Embedded Systems Essentials)

Issue 354 January 2020 | ISSN 1528-0608

CIRCUIT CELLAR® (ISSN 1528-0608) is published monthly by:

KCK Media Corp.
PO Box 417, Chase City, VA 23924

Periodical rates paid at Chase City, VA, and additional offices. One-year (12 issues) subscription rate US and possessions $50, Canada $65, Foreign/ ROW $75. All subscription orders payable in US funds only via Visa, MasterCard, international postal money order, or check drawn on US bank.

SUBSCRIPTION MANAGEMENT

Online Account Management: circuitcellar.com/account
Renew | Change Address/E-mail | Check Status

CUSTOMER SERVICE

E-mail: customerservice@circuitcellar.com

Phone: 434.533.0246

Mail: Circuit Cellar, PO Box 417, Chase City, VA 23924

Postmaster: Send address changes to
Circuit Cellar, PO Box 417, Chase City, VA 23924

NEW SUBSCRIPTIONS

circuitcellar.com/subscription

ADVERTISING

Contact: Hugh Heinsohn

Phone: 757-525-3677

Fax: 888-980-1303

E-mail: hheinsohn@circuitcellar.com
Advertising rates and terms available on request.

NEW PRODUCTS

E-mail: editor@circuitcellar.com

HEAD OFFICE

KCK Media Corp.
PO Box 417
Chase City, VA 23924
Phone: 434-533-0246

COPYRIGHT NOTICE

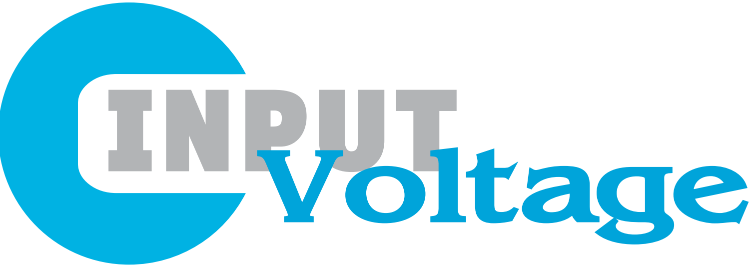
Entire contents copyright © 2019 by KCK Media Corp. All rights reserved. Circuit Cellar is a registered trademark of KCK Media Corp. Reproduction of this publication in whole or in part without written consent from KCK Media Corp. is prohibited.

DISCLAIMER

KCK Media Corp. makes no warranties and assumes no responsibility or liability of any kind for errors in these programs or schematics or for the consequences of any such errors printed in Circuit Cellar®. Furthermore, because of possible variation in the quality and condition of materials and workmanship of reader-assembled projects, KCK Media Corp. disclaims any responsibility for the safe and proper function of reader-assembled projects based upon or from plans, descriptions, or information published in Circuit Cellar®.

The information provided in Circuit Cellar® by KCK Media Corp. is for educational purposes. KCK Media Corp. makes no claims or warrants that readers have a right to build things based upon these ideas under patent or other relevant intellectual property law in their jurisdiction, or that readers have a right to construct or operate any of the devices described herein under the relevant patent or other intellectual property law of the reader's jurisdiction. The reader assumes any risk of infringement liability for constructing or operating such devices.

© KCK Media Corp. 2019 Printed in the United States



The Science of Technique

While I was an engineering student in college—back when dinosaurs roamed the Earth—I didn't have a lot of free elective slots. Because most of you *Circuit Cellar* readers are engineers, you can probably relate. But I did manage to fit in an elective course about the philosophy of technology. Whether it was the engineer in me or the nerd in me, I've always had an interest in "looking under the hood" or "behind the scenes" at the underlying meaning of things, and of words in particular. One thing that stuck with me from that course was the notion that the word technology when you break it down means "the science of technique." For me that phrase has a nice ring to it.

Particularly in the past decade or so—as technology has become a part of everyday consumer life—the shortcut term "tech" has emerged as a cool replacement for "technology." I've always bristled at that, especially as it became clear that the word "tech" tends to be used more frequently by those that don't have any clue about how electronic circuits and computing systems work. And by leaving off the "-ology" they are leaving off the "science of" part, which, to me, is the important bit.

The result of all this is that I've tended to be stubborn about not using the shortcut term "tech" in either writing or in conversation. That's easier said then done when trying to keep headlines short, and I've softened my stance about it in recent years. In an era when boosting website SEO requires a certain amount of conciseness, one must adapt.

Now that I've gotten that off my chest, I'll turn a technology (wink) that is definitely well positioned to be a key "under the hood" winner: RISC-V. As a free and open instruction set architecture, RISC-V has shaken things up in the processor realm by offering an ISA that everyone can use without paying a license fee. The RISC-V specification enables custom instruction extensions to facilitate the design of Domain-Specific Architecture/Acceleration (DSA). These are important for applications such as Artificial Intelligence/Machine Learning, AR/VR, ADAS and next generation storage and networking.

The timing of this magazine's production is such that I'm not able to report on the 2019 RISC-V Summit that took place in early December. Judging by progress made in the RISC-V ecosystem throughout 2019, I'm sure there were many interesting developments. Instead, I'll talk about the market trends in RISC-V. In November, Semico Research released a new report "RISC-V Market Analysis: The New Kid on the Block" that estimates that the market will consume a total of 62.4 billion RISC-V CPU cores by 2025, with the industrial sector forecasted to be the largest segment with 16.7 billion cores. Forecasting the compound annual growth rate (CAGR) for RISC-V CPU cores, Semico estimates that segments including the computer, consumer, communication, transportation and industrial markets will see a 146.2% percent CAGR on average between 2018 and 2025.

In its forecast of the CAGR for RISC-V CPU cores between 2018 and 2025, Semico estimates that the communication sector will see the largest CAGR due to the deployment of 5G and the multitude of products and applications that will be enabled with the adoption of 5G technology. Transportation is estimated to have the second-fastest CAGR due to the automotive industry's growing focus on electrification and the increased adoption of CPU-based systems for safety, in-cabin experiences, driver assistance and wireless communications. Semico not only found that organizations are designing RISC-V solutions across a variety of performance and volume applications, but also that they're designing anywhere from one or two to more than 1,000 cores in SoCs.

RISC-V is compelling technology for engineers to design into products—products that end customers are free to call "tech" if they so choose.

Jeff Child

COLUMNS

46 DATASHEET
COM Express Boards
Compact Performance
By Jeff Child

50 Embedded System Essentials
Building Against Fault Injection Attacks
Cautious Coding
By Colin O'Flynn

54 Picking Up Mixed Signals
Relaxation Generator: Reloaded
Internet Era Upgrade
By Brian Millier

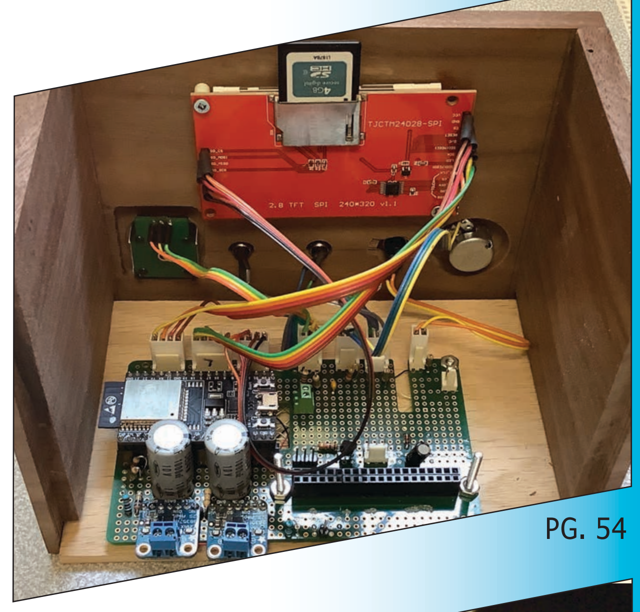
64 The Consummate Engineer
Semiconductor Fundamentals
(Part 5) More on FETs
By George Novacek

68 From the Bench
Shedding Light on Smart LED Design (Part 1)
Programming and Pixels
By Jeff Bachiochi

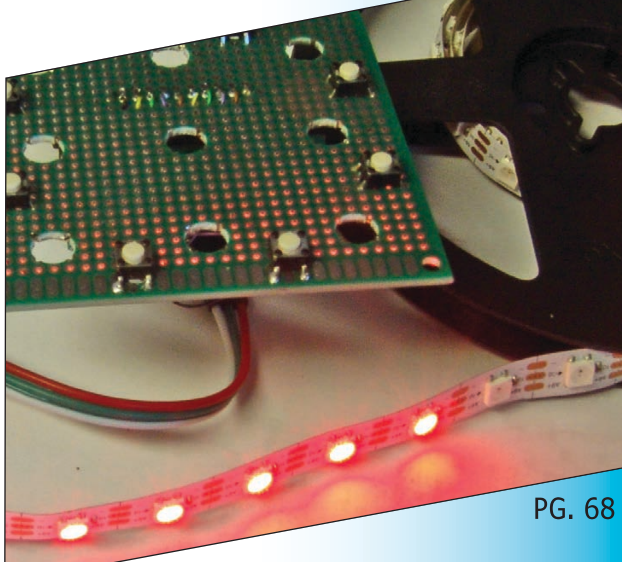
79 TECH THE FUTURE
The Future of IoT as Safety Resource
Safer Living Through AI and IoT
By Jen Bernier-Santarini

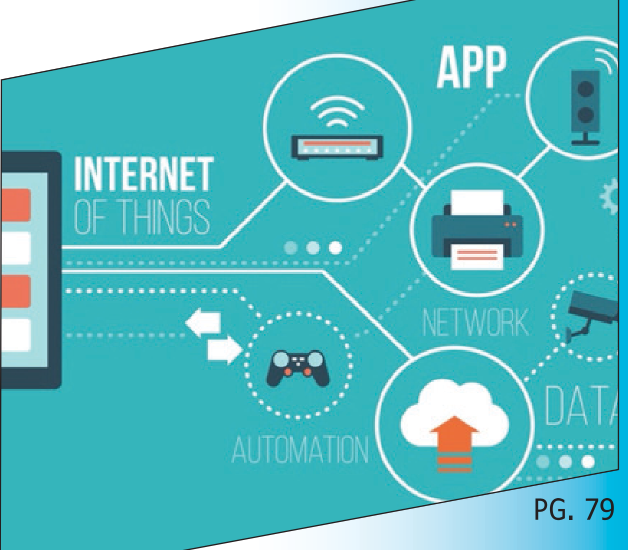
76 : PRODUCT NEWS
78 : TEST YOUR EQ

@editor_cc
@circuitcellar
circuitcellar

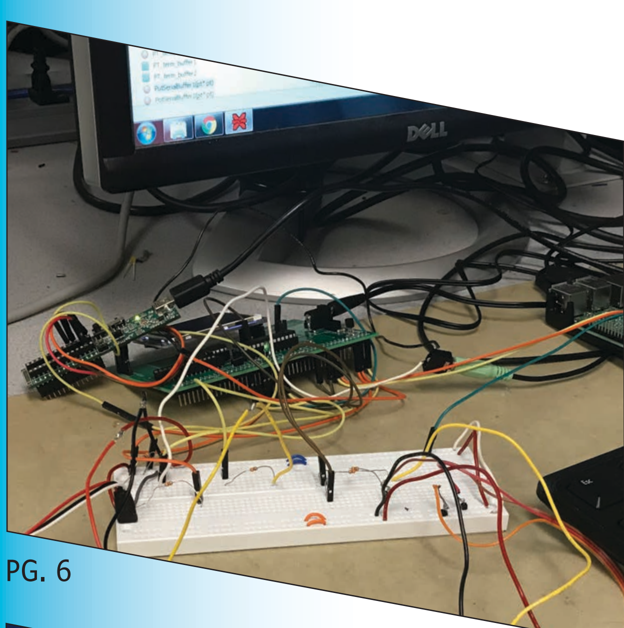


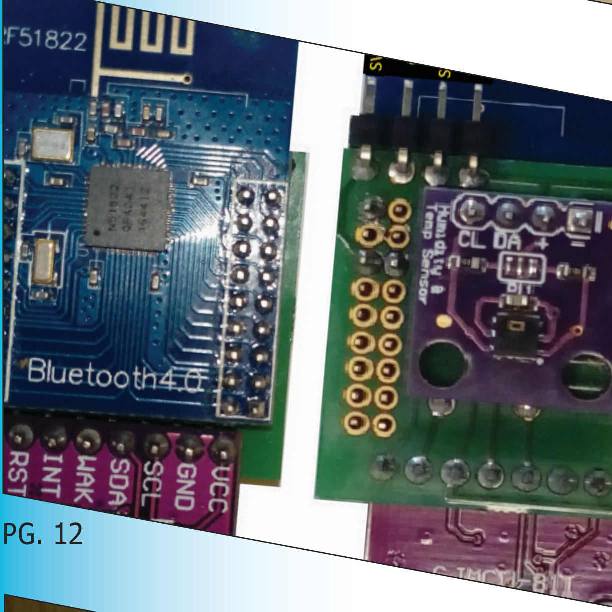
PG. 54


PG. 68

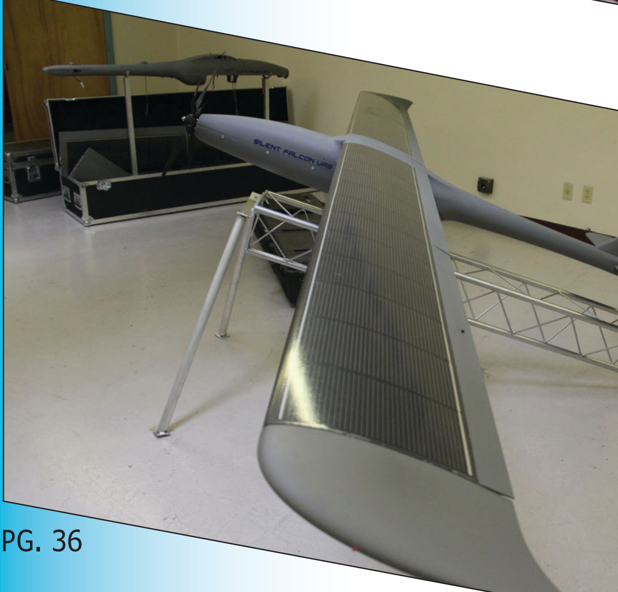


PG. 79


PG. 6



PG. 12


PG. 36

6 **Motion/Gesture-Controlled Speakers**
PIC32 Playback
By Jidenna Nwosu, Benjamin Francis and Ayomi Sanni

12 **Device Measures Indoor Air Quality**
Bluetooth-Based Design
By Carlo Tauraso

20 **Sound Localization**
Using a PIC32 MCU
By JinJie Chen and Alvin Pan

27 **Choosing Real-Time Embedded System Products**
10 Key Tips
By Rodger Hosking

32 **System Controller Manufacturing Test (Part 1)**
The Hardware
By Nishant Mittal and Manoj Khandelwal

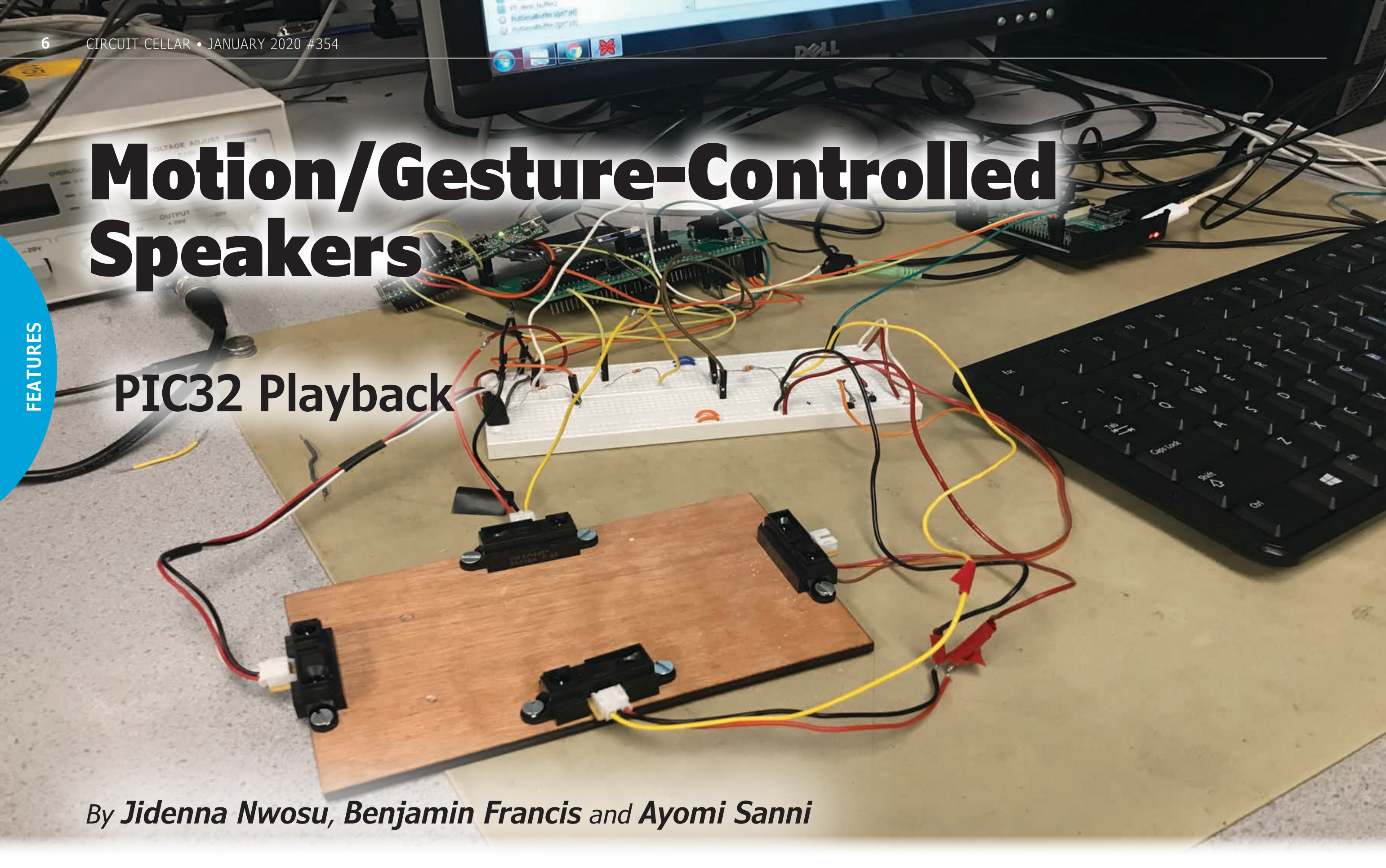
SPECIAL FEATURE
36 **System Solutions Accelerate Drone Development**
Fast Track to Flight
By Jeff Child

TECHNOLOGY SPOTLIGHT
42 **Analog ICs Boast Battery Management Innovations**
Perfecting Power
By Jeff Child

FEATURES

Motion/Gesture-Controlled Speakers

PIC32 Playback



By *Jidenna Nwosu*, *Benjamin Francis* and *Ayomi Sanni*

Controlling electronic devices with hand gestures may seem like the stuff of science fiction. But the technology is easily available today, even for MCU-level embedded systems. Learn how these three Cornell students built a motion/gesture-controlled speaker using sensors, a computer and a Microchip PIC32 MCU. With hand gestures, the system lets you control the volume, play/pause and change songs by skipping forward and backward.

Our Motion-Controlled Speaker project is an application that uses non-contact sensors to control the audio output from a speaker, based on motion patterns that the sensors detect. This project idea originated when we began discussing innovations that would be of interest to us. We immediately took a liking to this idea, because we could see it being implemented into products in the near future. We all have personal interests in music and in working on something that could be built into different products, such as smart watches or other similar smart products with streaming capabilities. Our objective was to build a prototype of this type of technology, using Sharp GP2Y0A21YK0F IR sensors from Pololu, the PIC32 microcontroller (MCU) and another main component that would be based on whether we decided to stream the music or to play downloaded music from a device with memory.

After much research and trying out various methods, we decided to use a Raspberry Pi 3B embedded computer board as a device for the playback of songs. The final product, using the IR sensors, PIC32 and Raspberry Pi 3, was a working prototype that was able to pause and play songs, turn the volume up and down and change to the previous and next songs—solely based on hand motions. The schematic of the project is shown in **Figure 1**.

HIGH-LEVEL DESIGN

A significant logical part of our project was the communication between the Raspberry Pi and the PIC32 MCU, through the utilization of the UART hardware on each device. The serial port between the two was set up at a baud rate of 115200 bps—the fastest speed that the serial port can transfer data between the two computers. There was an optimization trade-off that we had to consider. We knew that using this baud rate allowed us to send the greatest amount of data at a fast enough rate, but with a higher chance of data corruption or data loss. Fortunately, we didn't observe any of these errors, so we chose to continue using the highest speed possible.

The baud rate is the speed at which bits

can be transferred, so bytes of data can be sent at a maximum rate of 11,500Hz. With that in mind, we had to downsample most tracks of music, which originally were sampled at a rate of 44,100Hz. We used Mathworks MATLAB code provided by a Cornell professor to downsample the tracks to a quarter of that frequency, or 11,025Hz, which was the greatest rate we could obtain that was below the maximum. This decreased the quality of the relayed music, but it was still clear and enjoyable.

The structure of our project is as follows: The first part is resetting the PIC and running our code on the Raspberry Pi 3. Once the PIC has finished resetting, it sends a ready signal to the Pi. The Pi receives this signal and then begins to process and send the music data that has been prewritten onto it, byte by byte. These bytes are received by the PIC and stored into two buffers, where one receives the data, and when full, starts playing. While this buffer is playing, the other buffer continues receiving data where the previous left off. When one of the buffers is full, it sends a signal that it is ready to receive more data. **Figure 2** is a logical structure diagram illustrating this process.

We made sure to follow the typical multi-processor communication protocol that was relevant for our purposes. Because we were using the UART hardware present on the PIC and the Pi, we followed the RS-232 standard. Unlike other motion-activated audio emitters, our project is more focused around using specific motions to control a sound or music playback device, with different patterns of motion producing different results.

HARDWARE DESIGN

The PIC32 does not have enough memory (only 128KB of flash memory) to store full

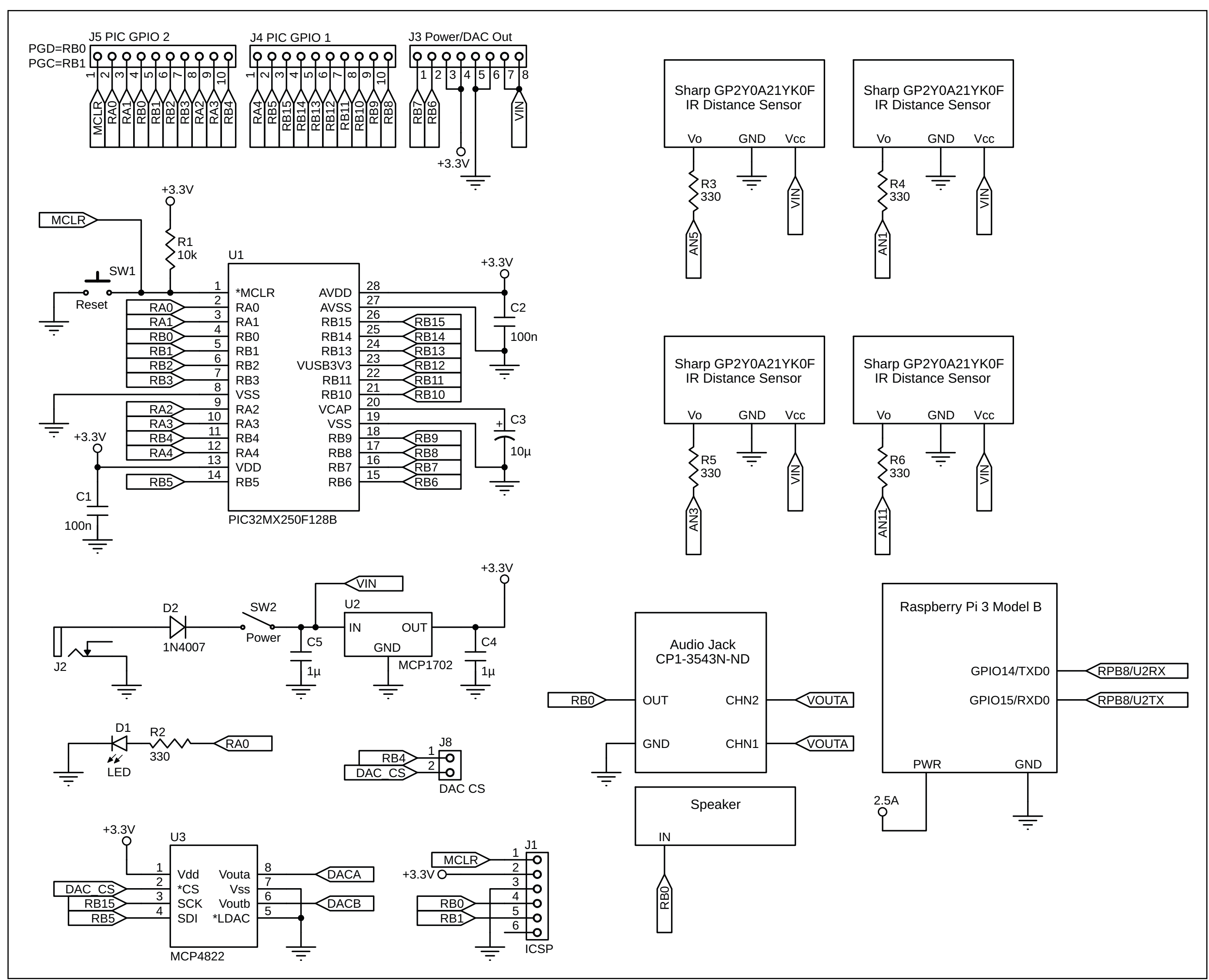


FIGURE 1

Schematic for our Motion Sensor Speaker system

FEATURES

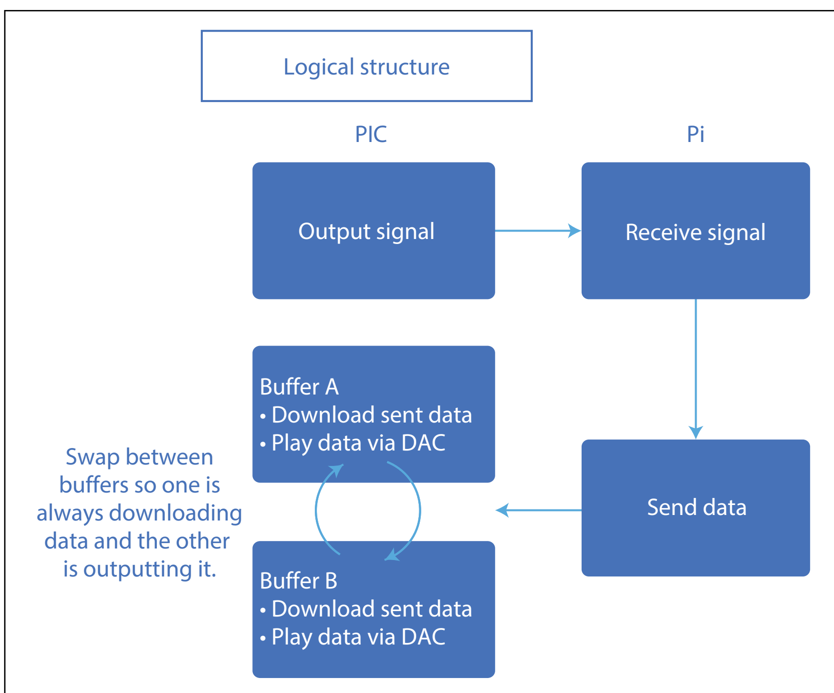


FIGURE 2
Logical structure diagram

songs. So, we used the Raspberry Pi 3 to store music and stream to the PIC32, because its memory is limited only by the size of its memory card, and it is also a high-performance device for its price. The serial communication between the PIC and the Raspberry Pi required connecting the UART transmit pin on the PIC to the UART receive pin on the Raspberry Pi and vice versa. We also ensured that they share a common ground.

This sensor array acts as the interface between a user and the music streaming system that we designed. We screwed the sensors to a small rectangular wooden board to keep them stable and make it easy to control the program using hand gestures. We assembled the sensors in a diamond formation (**Figure 3)**. This formation makes it easy to control the flow of the music by simply holding a hand over different combinations of sensors to perform different actions. For example, holding a hand over the bottom and top sensor pauses or resumes playing the music. The diamond formation also makes it possible to add a swiping feature to our system in the future, such that swiping across the sensors from left to right will switch the music to the next track.

PROGRAM DESIGN

The main software component of our motion-sensor speaker system consists of two threads: an interrupt service routine (ISR) on the PIC and a Python program on the Raspberry Pi. The program continuously executes until it is terminated. For the entire system to run successfully, the threads, ISR and Raspberry Pi program must be synchronized with each other and communicate efficiently. The sensor thread reads the analog input from the analog IR distance sensors and controls the state of the system based on these data.

The serial thread's main function is to spawn another thread that reads and sends data through the UART module based on the state of the system. The ISR processes data received through the UART and outputs the processed data through the digital-to-analog converter (DAC). The serial communication program that runs on the Raspberry Pi loads the music header files and sends these data through the UART. This program also receives data through the UART from the PIC that affect the state of the program.

For testing, we used a MATLAB program to make WAV files. This program converts downloaded WAV files to C language header files that can be outputted through the DAC once transmitted to the PIC. This program first reads a WAV audio file specified at a certain location on the computer. The WAV files have a sampling frequency of 44.1kHz, which is too fast for our system to play, so the program down-samples the audio by a factor of four.

This allows the audio to be played at a sampling frequency of about 11kHz. The samples then have to be scaled so they can be played by the 12-bit DAC. These converted audio samples are then stored to the header file, with enters between samples. After being converted to a header file, the music is ready to be loaded into our program to be played.

PIC32 SECTION

The first PIC32 thread begins by reading the first four channels of analog-to-digital converter (ADC). The ADC converts the analog output from the four IR distance sensors to a digital format that is then stored in variables (`adc_9`, `adc_10`, `adc_11`, `adc_12`). We set a minimum threshold of 400 ADC units for a sensor reading that counts as a valid detection.

ABOUT THE AUTHORS

Jidenna Nwosu is a Cornell University graduate who majored in Electrical and Computer Engineering and Information Science Engineering. Jidenna is looking to work as an embedded software/hardware engineer.

Benjamin Francis is a Cornell University graduate (May 2019) who majored in Electrical and Computer Engineering. He is currently work as a Systems Engineer at L3Harris Technologies.

Ayomi Sanni is a Cornell University graduate (May 2019) who majored in Electrical and Computer Engineering. Ayomi is currently looking to work as a software engineer.

We found this to be an ideal threshold through trial and error. If the threshold is too small, then you have to hold your hand too close to the sensor for motion to be detected. If it's too large, then objects that are far away may be unintentionally detected.

We implemented a counter for each sensor to keep track of how long a hand is being detected by a sensor. The corresponding counter is incremented with every consecutive iteration of this thread during which a hand is still being detected by the same sensor. If a sensor no longer detects a hand, then its corresponding counter is reset to zero. These counters are a form of debouncing the sensors. For example, if someone quickly waves a hand over a sensor by accident, it will not be acknowledged by our program.

These counters act as the control signal for the state of the music playback. They also signal actions that should be done to the playback. The two states that our system can be in are "play" and "pause." When someone holds a hand over the bottom and top sensors, one state is switched to the other state. In other words, the state switches from play to pause or pause to play. We found that if the top counter is equal to 3 and the bottom counter is greater than 1, it is a solid enough sign that someone is attempting either to resume playing music or pause the music. We discovered that when we set the condition for both counters to be equal to the same value, the switch of states was inconsistent.

The volume of the music can be turned up or down when a hand is held over the top or bottom sensor, respectively. To adjust the volume, the counter corresponding to the sensor must be equal to 2, and the sensor opposite it must be less than 1. We added the "less than 1" condition to differentiate this action from changing the play/pause state. A variable for volume is then decremented/incremented based on which action is signaled.

We also implemented an action state for switching to a new song. When a hand is held over the right or left sensor, the next/previous track should play. This thread sets the action variable to next or previous track if either of these counters is equal to 2. The desire to switch tracks is later signaled to the Raspberry Pi by the serial thread.

SECOND THREAD

The second PIC thread spawns another thread that communicates with the Raspberry Pi by sending and receiving data through the UART's Tx and Rx pins. Each byte of data is received by the PIC and stored into one of the buffers, while the other one is being read in the ISR. We use two buffers to ensure constant playback, since one buffer constantly receives the data while the other relays that data to the DAC. This is more efficient and necessary so that data can be received and written at the same time that data are being played. It enables a seamless transition between bytes of music data that are transferred. The spawned thread is not killed until the current buffer being written to has been filled with the latest 8,000 samples of the transmitted music data.

In the spawned thread, we continuously check the state of our system and send a signal to the Pi based on this state. When the state of our system is in "play," we continuously receive data from the Pi by first sending it the ready signal. And when the state is in "pause," we stop sending the ready signal, which stops the transfer of data—but we make sure to save the spot that we stopped at on both ends. When switching to the next or previous track, we output the corresponding signal that basically informs the Pi to begin outputting data from the next or previous set of data that was downloaded. Each set is given a corresponding number on the Pi side.

The next or previous signal is sent only once, then we return to continuously sending the ready signal, so that the PIC, in turn, receives and plays the song immediately. This makes the entire song-switching process occur in real time. It is important to clear either buffer when necessary, such as when switching songs, so that playback of different song data does not overlap—an issue that we encountered briefly.

INTERRUPT SERVICE ROUTINE

The ISR formats the samples in the serial

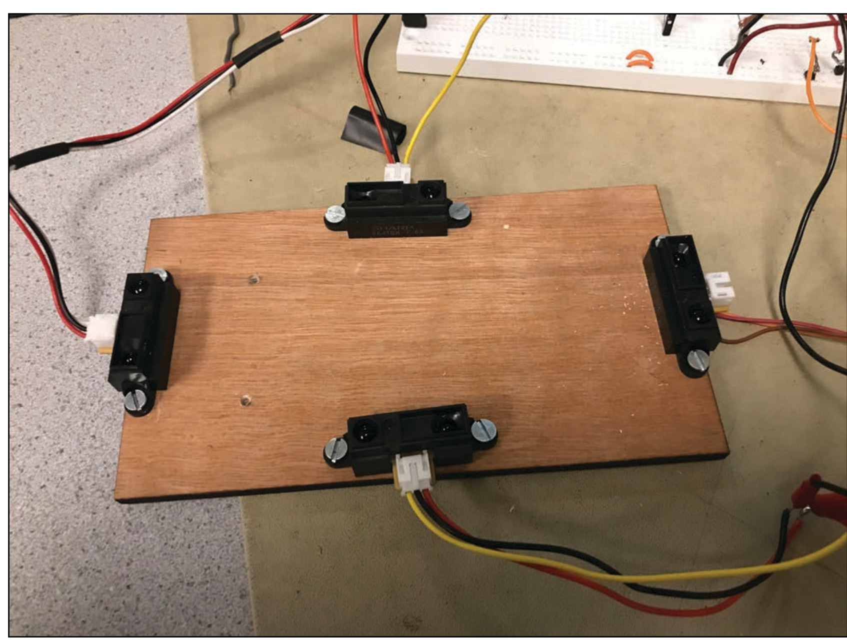


FIGURE 3
Sensor board with diamond formation of sensors

FEATURES

FIGURE 4
QR code for YouTube video demonstration of our project. The video is also posted on *Circuit Cellar's* article materials webpage.

buffers for the DAC, before transmitting them through the second SPI channel and outputting through the DAC. This ISR is triggered by a timer interrupt at a rate determined by the sampling frequency of the music playback. We set the timer to trigger an interrupt every 3,628 cycles (1/11,025Hz) (PIC32 clock freq./sampling frequency). Each time the ISR is executed, a new sample is sent through the SPI channel to the DAC, only if the state of the system is in "play." Before the sample is transmitted through the SPI port, the sample is manipulated based on the state of the system and requirements for the DAC. The sample is first converted to an integer, before being left-shifted by a number determined by the volume variable.

A greater left shift creates a larger value, which consequently makes the sample louder. The most that each sample can be left-shifted is four. That's because the samples are 8-bit values, and the DAC only supports 12 bits. The samples cannot be left-shifted by less than zero, because that would result in the loss of some of the sample's data. The shifted sample is then added to 2,048 to increase its amplitude, to maximize the potential of the DAC. Before being written to the SPI channel, the sample is OR-ed with the DAC A configuration bits. This step tells the SPI to send the sample to DAC A.

The buffer that is not being written to at the time is read and sent through SPI to the DAC. Once this buffer's samples have all been transmitted to the DAC, the ISR switches the buffer state. This indicates that this buffer should now be written to, and the other buffer should be read from. This switching of buffers allows for continuous playing of music, because samples from the Pi are always being received and stored at the same time that the PIC is outputting these samples.

Pi SECTION

This program's main purpose is to transmit music samples to the PIC using serial communication. It reads and writes serial data through the UART module on the Raspberry Pi. The program begins by initializing a serial writer and reader, to send and read signals through the UART. We set the baud rate for this communication to 115,200bps—fast enough to transmit 11.5 thousand samples per second). We then read the header files into variables, which we convert to integer format. One final conversion is then performed on the data: conversion to byte arrays format. At this point, the music samples can be transmitted through the Raspberry Pi's UART transmit pin.

Once the initial procedures performed on the data are completed, the program enters an infinite loop and begins reading the serial input. If an "A" is received, then a flag is set to indicate that the PIC is waiting to receive data. Once this flag has been set, the next 8,000 samples of the current song being played are transmitted through the Pi's transmit pin to the PIC.

If the song is finished being transmitted, then on the next iteration, the next song starts being transmitted to the PIC. If an "N" (or "P") is received instead, the next (or previous) song read by the program starts to be transmitted. Our current Python serial communication program transmits only three songs, but this program can be expanded to transmit many more songs simply by converting and loading more WAV files onto the Raspberry Pi's memory. It will also require duplicating much of the code.

RESULT OF DESIGN

We were pleased with what were able to achieve with our project. It covered all the bases of what we had initially aimed to do. When we loaded three songs through the Pi, we were able to pause and play a song, skip forward to the next song, skip back to the previous song and control the volume. The Pi read properly from the serial interface, and did not start transmitting the music until it received permission from the PIC. The PIC read the inputs from the distance sensors, and used that information either to control the volume level, or tell the Raspberry Pi to stop transmitting or change the song it was sending. All of this was performed quickly and smoothly, and—most importantly—in real time. We were able to exhibit all of this during our demo. To see a YouTube video of our project demo, scan the QR code in **Figure 4**. This video is also posted on *Circuit Cellar's* article materials webpage.

The design also showcases all the things we considered throughout the development of the speaker. By positioning the sensors on a board similar to a remote and in an efficient manner, we ensured that each gesture will be correctly interpreted. This design as a whole is preferable and useful because all it takes is a simple hand gesture over the apparatus to control the user's music.

For detailed article references and additional resources go to:
www.circuitcellar.com/article-materials

RESOURCES

Digi-Key | www.digikey.com

Mathworks | www.mathworks.com

Microchip Technology | www.microchip.com

Pololu | www.pololu.com

A more advanced version of this prototype could be useful in many situations. The primary and most popular use would be providing a fun, innovative and relatively effortless method in which users can interface with their devices. However, there are also some serious applications for this technology. For example, people with impaired vision could benefit greatly from this type of gesture technology. If the motion sensing speaker were attached to the user's wrist—perhaps as part of a smart watch application—the user could switch between songs or perhaps pages of an audio book without needing touchscreen buttons. The user would simply wave a hand over the screen in the desired direction to switch or flip pages. Those with mental disabilities could possibly benefit from this technology as well, because they might find it easier to use certain gestures and hand motions, rather than the typical button inputs required to interface with devices. Overall, we can see many amusing and functional uses for a more advanced version of this prototype.

CONCLUSIONS

On the whole, our final product worked quite seamlessly and met, if not exceeded, our expectations. We had to deviate from some initial plans as we progressed with this project. But, in the end, we fully achieved our goal of having a quality speaker system that could be motion-controlled by hand gestures.

There are a few supplements we could add in the future to further bolster our system. It would be desirable to develop a method of streaming the audio WAV file directly on the Raspberry Pi. This would preclude the lengthy process of the MATLAB header conversion for each song. Another improvement would be refining the gesture-detection software, so that users could perform more engaging motions, such as swiping up to raise the volume, and swiping down to lower it.

An additional interesting feature that we talked about was installing a microphone that could "listen to the room." It would adjust the volume of the music based on the background noise present in the current environment. However, this might not be feasible or desirable in some situations. Our final, most advantageous improvement would be the ability to play music from a streaming application, such as Spotify or Apple music. This would make our system a lot more useful and popular. All in all, these improvements would be nice additions to our system, but we are very happy with our current final product.

Device Measures Indoor Air Quality

Bluetooth-Based Design

By
Carlo Tauraso

Unhealthy air in indoor environments has been linked to diseases such as asthma, allergies, lung infections and more. That's driven the demand for sophisticated indoor air quality (IAQ) measurement systems, but many have serious limitations. In this project article, Carlo shares the details of his design of IAQnet project. The Bluetooth-based system creates a network of IAQ monitoring tags that enables users to evaluate the healthiness of multiple environments.

Air pollution has become a problem that cannot be underestimated, due to its implications in the increasingly frequent catastrophic events of which we are powerless spectators. In recent years, governments have tried to limit not only global emissions but also possible sources of pollution in the buildings where we live.

Many researchers have shown a strong correlation between exposure to pollutants in indoor environments and some widespread diseases, such as asthma, allergies, lung infections, some forms of cancer and diseases affecting the cardiovascular system. Terms such as SBS (Sick Building Syndrome) and THS (Toxic Home Syndrome) have been coined to highlight and group all those symptoms of health deterioration of occupants in environments where there is polluted air. It has been calculated that people spend about 90% of their time indoors, in places such as schools, offices and apartments, so the health impact of the air we breathe in these environments is much greater than that resulting from outdoor air pollution.

On the market we have seen, in recent years, a rapid spread of air quality monitoring systems, especially those integrated into ventilation systems. Many of these are limited to measuring the indoor air quality (IAQ) in a single room and near the ventilation system.

Setting out to create an educational application that involves Bluetooth LE (BLE) technology, I thought of combining business with pleasure by developing a network of IAQ monitoring tags that allows evaluating the healthiness of multiple environments and one that is very simple to install—taking advantage of the Android smartphone features. This is how IAQnet was born—a small system consisting of one or more monitoring tags based on Bluetooth technology and an Android app communicating with them, displaying the values of some sources of pollution present in our house.

INDOOR AIR QUALITY AND SENSORS

Indoor air quality can be influenced by various kinds of contaminants, and currently there is no standard measurement method. One of the most promising methodologies is monitoring the levels of VOC (volatile organic compounds) and carbon dioxide (CO_2). VOCs are compounding whose toxicity depends on their density in the air we breathe.

The VOCs include: benzene, which is generated in the production of plastic materials; chlorofluorocarbons (CFCs), which are present in cleaning products and coolants;

350 ppm	1,000 ppm	4,000 ppm	5,000 ppm	50,000 ppm	100,000 ppm	200,000 ppm
Outdoor air	Feeling of stale air	Room with poor ventilation	Max concentration in the work station	Man expiratory concentration	Candle extinguishing	Deadly to man

FIGURE 1
Association of the CO_2 concentration in indoor air with the need for ventilation

methylene chloride, which is present in adhesives and spray paints; formaldehyde, which is present in plastic laminates on wood; acetone, which is found in many paints; and numerous other chemicals.

For this project I used a low-cost breakout board with a CCS811 sensor from AMS AG [1]. It's an ultra-low-power digital sensor with an I²C interface that integrates an MOx (Metal Oxide) gas sensor, to detect a wide range of VOCs and to predict TVOC (total volatile organic compounds). It includes a microcontroller (MCU) that uses an algorithm to process the values measured by presenting a TVOC value at the output, and then converts it into the equivalent CO_2 level. The TVOC output range is from 0 to 1,187ppb.

Clearly this equivalent level of CO_2 is not a direct measure of the CO_2 present in the environment, but rather the result of an equation application. Therefore, the sensor provides TVOC concentrations and an estimation of CO_2 or "eCO_2." The eCO_2 output range is from 400 to 8,192ppm. TVOC measurement is more important than CO_2 in terms of health impact. However, the equivalent CO_2 makes it possible to add the sensor output to ventilation standards and implement it for ventilation systems, thus reducing the energy consumption compared to time-scheduled ventilation. The equivalent CO_2 allows the detected TVOC value to be interpreted more clearly, through the use of tables (**Figure 1**) that associate the concentration of CO_2 with the need for ventilation. For example, to have a healthy environment in a room, the concentration of CO_2 should not exceed 1,000ppm.

For more precision, it is possible to compensate gas readings with variations in

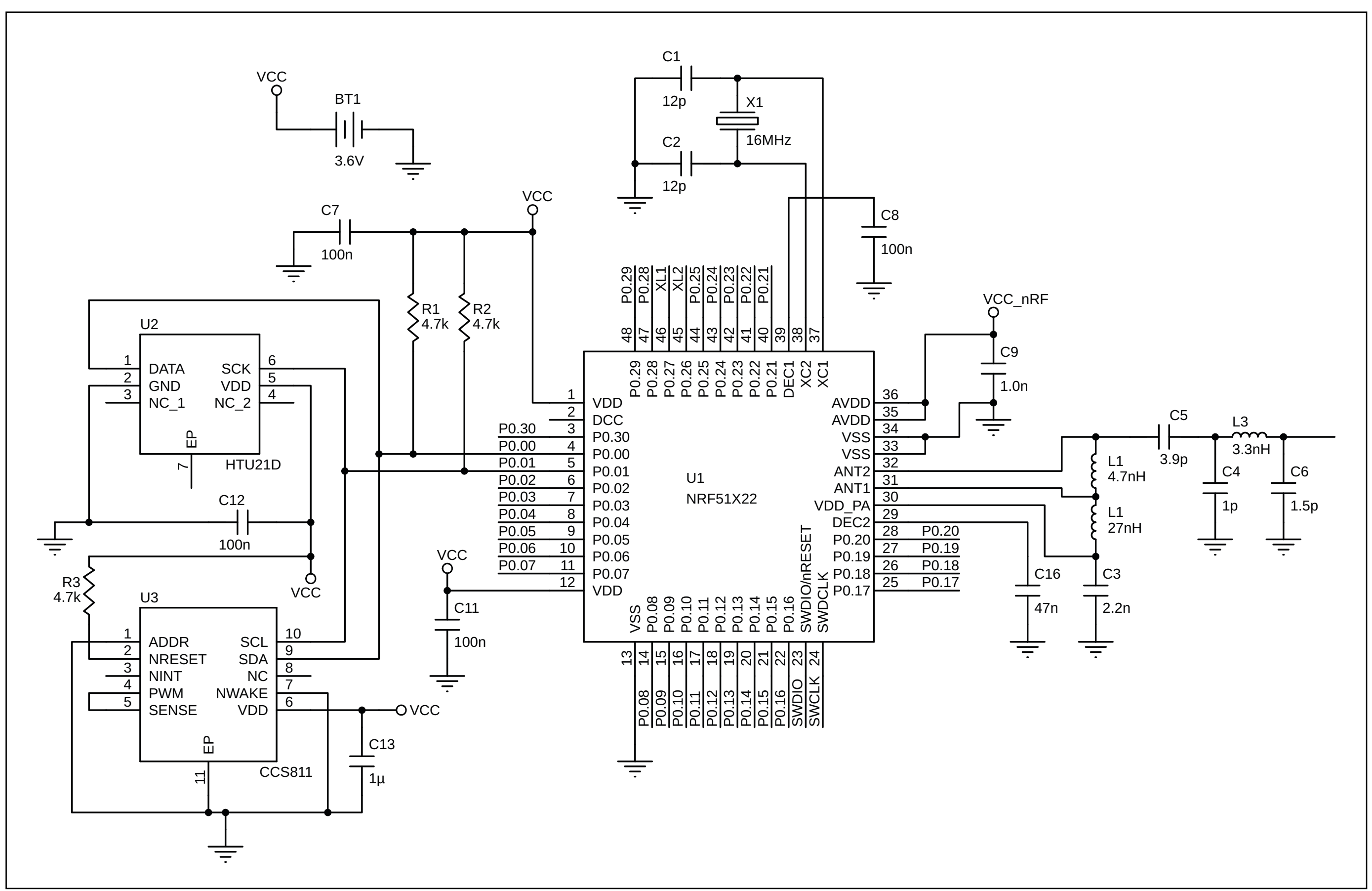


FIGURE 2
The IAQnet tag schematic

FEATURES

TABLE 1
Parts list for the IAQnet project

Parts List	
C1, C2	12pF capacitors
C3	2.2nF capacitor
C4	1pF capacitor
C5	3.9pF capacitor
C6	1.5pF capacitor
C7, C8, C11, C12	100nF capacitors
C9	1nF capacitor
C10	47nF capacitor
C13	1µF capacitor
R1, R2, R3	4.7kΩ resistors
X1	16MHz crystal
L1	4.7nH inductor
L2	27nH inductor
L3	3.3nH inductor
U1	nRF51822
U2	HTU21D
U3	CCS811

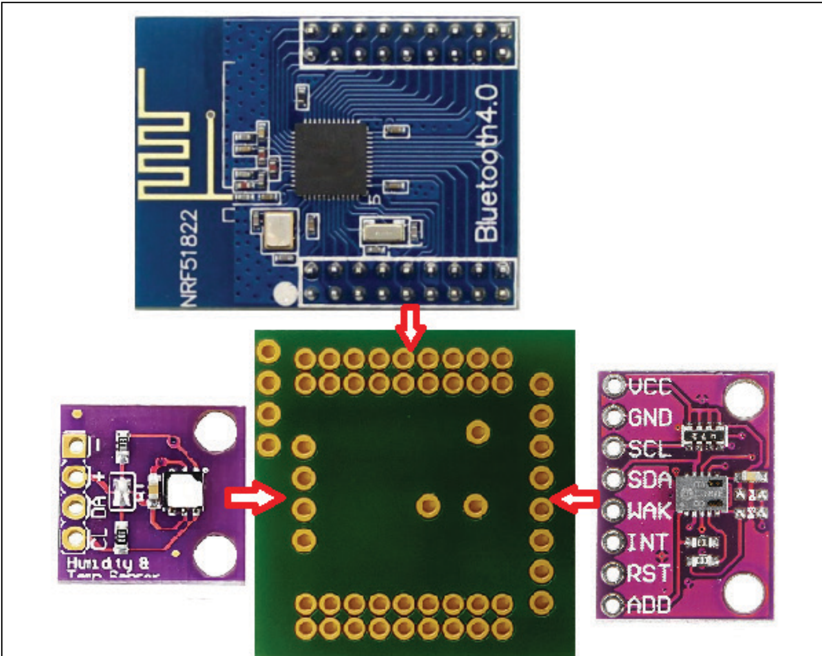


FIGURE 3
The tag assembly

ABOUT THE AUTHOR

Carlo Tauraso (carlotauraso@gmail.com) studied computer engineering at the University of Trieste in Italy and wrote his first assembler code for the Sinclair Research ZX Spectrum. He is currently a senior software engineer, who does firmware development on network devices and various types of micro-interfaces for a variety of European companies. Several of Carlo's articles and programming courses about Microchip Technology PIC MCUs (USB-PIC, CAN bus PIC, SD CARD, C18) have been published in Italy, France and Spain. In his spare time, Carlo enjoys playing with radio scanners and homemade metal detectors.

temperature and humidity. I therefore added another low-cost breakout board with an HTU21D sensor from Measurement Specialties [2]. It is a highly accurate temperature and relative humidity sensor. It also uses an I²C interface, so I share the same bus used for the CCS811. Default resolution is set to 12 bits relative humidity and 14-bit temperature readings, and is more than sufficient for our purposes. Measured data is transferred in 2-byte packages, MSB first. But the measured values require a conversion carried out by applying the formulas indicated on page 15 of the HTU21D datasheet [3].

DESIGNING THE IAQnet TAG

As shown in the schematic (**Figure 2**), the circuit consists of a core module based on Nordic Semiconductors' nRF51822 SoC and two breakout boards—one for detecting temperature/humidity, and one for detecting the concentration of VOCs. The nRF51822 is a multiprotocol SoC for ULP wireless applications [4]. It incorporates an Arm Cortex M0 CPU, 256KB flash memory, 32KB RAM memory and a powerful radio transceiver. The nRF51 series RF transceiver is interoperable with BLE (Bluetooth low energy) and other 2.4GHz protocol implementations such as ANT, Gazelle and others. In this project I used BLE to develop the Android app. This makes the app for reading the data detected by the tag simpler and more affordable, even for less experienced readers. However, the system also allows the implementation of an ad hoc communications protocol that is also fully air-compatible with the nRF24L series that I used in a thermal monitoring system project published a few years ago entitled " Build a Thermal Monitoring Network" (*Circuit Cellar* 288, July 2014).

For the prototype, I used another breakout board from Waveshare [5], with the nrf51822 in the basic configuration (**Figure 2**). The oscillator circuit consists of a crystal at 16MHz with two capacitors C1 and C2. The capacitors C9 and C11 have decoupling function on their power source pins. Now, let's look at the antenna section. For space reasons I have used the model with an integrated PCB antenna. The circuit has an impedance network adapter with capacitors and inductors (L1, L2, L3, C3, C4, C5 and C6). Adapting the impedance is critical to avoid losing power. **Table 1** shows the parts list.

Pins P0.00 and P0.01 are configured, respectively, as data line (SDA) and clock line (SCL) of the I²C communication bus between the nrf51822 and the HTU21D/CCS811. R1 and R2 are pull-up resistors. Note that both breakout boards contain two SMD pull-up resistors. On the CS811 board they are 4.7kΩ

(472SMD), and on the HTU21D they are 10kΩ (103SMD). To avoid connecting the two pairs in parallel—reaching a total resistance that is too low—there are pads on the board that allow you to connect or disconnect the integrated pull-up resistors. In this project, I used only the 4.7kΩ resistors present on the HTU21D board.

The CCS811 chip operates in polling mode. A measurement is performed every second (DRIVE_MODE = 001). The host software cyclically reads data from the sensor, performing a 4-byte data read to the register named ALG_RESULT_DATA. Each pair of values should be converted to a 16-bit type field value. In this way it is possible to obtain the eCO2 and TVOC values directly. The CCS811 supports compensation for relative humidity and ambient temperature. So, before every TVOC reading, it is possible to update ENV_DATA registers with temperature and humidity values from HTU21D.

In power-sensitive applications, the WAKE pin is controlled by a GPIO pin. In my project, I tie it to ground, so the chip never enters sleep mode. The ADDR pin is low, so I^2C transactions use the 7-bit address 0x5A—which is different from the address used by the HTU21D (0x80), because the I^2C bus is shared. The RESET pin is an active low input, and is pulled up to VCC by default, so I connect an external 4.7kΩ pull-up resistance (R3) to avoid erroneous resets. It is also worth considering that the CCS811 sensor has a 20-minute condition period before accurate readings are generated. Furthermore, the manufacturer AMS advises customers to run the CCS811 for 48 hours, because the performance in terms of sensitivity changes during early use.

To ensure a simple and very small assembly, I created an interconnection layer. It is a small card that allows you to simply connect the three cards without having to add any other components. In **Figure 3**, the green small card is in the center. The interconnection board also includes a connection strip for a battery, and the JTAG bus (VCC, SWDIO, SWCLK, GND) to update the firmware. Finally, for debugging purposes, I have included some messages in the firmware during the initialization and measurement phases. They are sent through a serial interface that uses pins P0.05 RX, P0.06 TX, P0.07 CTS and P0.12 RTS. If you connect a TTL-to-RS232 converter to these pins, you can view information in terminal as:38,400bps/8/none/1/no flow ctrl.

In **Figure 4**, you can see the top and bottom copper layers of the interconnection layer. As you can see, it is a small card with a very simple scheme to connect the three breakout boards together. Assembly takes place by welding the cards to the interconnecting layer, one above the other, like a sandwich. In **Figure 5** you can see the assembled prototype board.

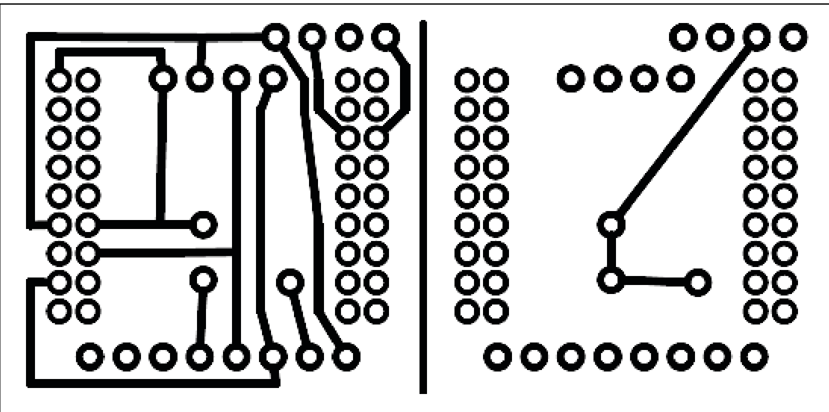


FIGURE 4
The interconnection layer

THE FIRMWARE

The firmware that runs on the core module consists of two parts: a BLE protocol stack (S110) and a user application. The nRF51 series SoCs are programmable with software stacks available from Nordic Semiconductors. These stacks are known as SoftDevice. They make application development flexible, so we can concentrate on the logic of the user application. Moreover, it is possible to integrate the same hardware on platforms that use other protocols by replacing the SoftDevice with the appropriate one. In this project, the user application acquires the values of the two sensors, and makes them available to an Android app via Bluetooth by

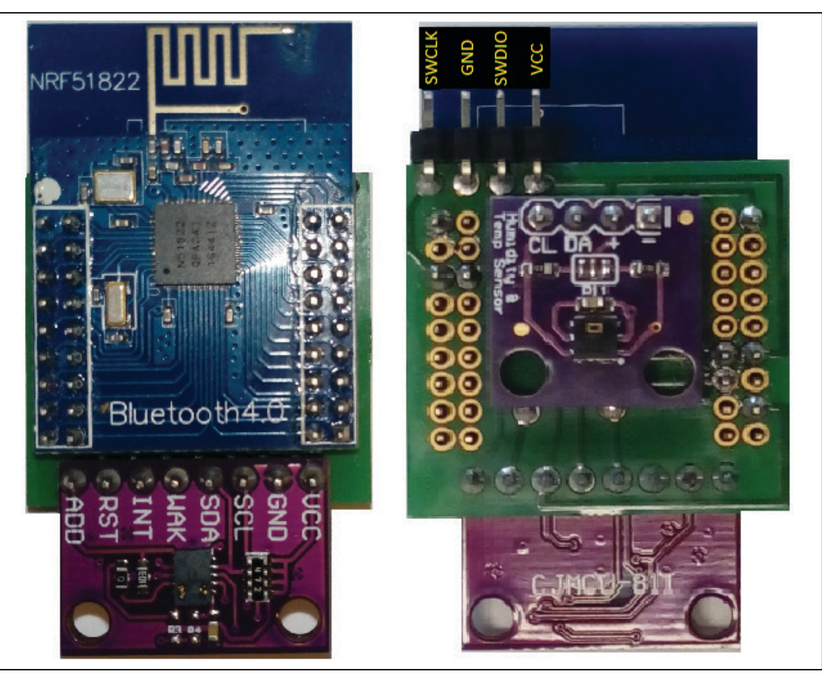


FIGURE 5
The assembled prototype board

FEATURES

```
INITIALIZE AND LOAD BLE STACK
INITIALIZE GAP PARAMETERS [device name, connection interval]
INITIALIZE ADVERTISING PARAMETERS [advertising interval, timeout]
INITIALIZE CUSTOM SERVICE PARAMETERS [assign UUID, create characteristics, assign permissions]
INITIALIZE UART [only for debugging purposes]
INITIALIZE I2C BUS

HTU21D SOFTRESET
HTU21D CONFIGURATION [Resolutions: Temperature 12bits, Rel. Humidity 14bits]

CCS811 read HW ID [reading 0x81 for CCS811]
CCS811 read STATUS
IF APP_VALID=1 [a valid firmware image is present]
START SENSOR FIRMWARE [transition from boot mode to firmware mode]
CCS811 read STATUS
IF FW_MODE=1 [the sensor firmware is ready]
CCS811 CONFIGURATION MODE1 [measurement every second and interrupts disabled]
START ADVERTISING [to be found by a mobile device]
CUSTOM SERVICE LISTENING FOR CONNECTIONS SHARING TWO CHARACTERISTICS
MAIN LOOP
START TEMPERATURE MEASUREMENT
START REL HUMIDITY MEASUREMENT
CONVERT TEMP AND REL HUM VALUE
WRITE TEMP AND HUM IN CCS811 ENVIRONMENT DATA REGISTRY
WHEN READY READ ECO2 AND VCO
UPDATE CHARACTERISTICS IN CUSTOM SERVICE
END MAIN LOOP
```

LISTING 1
The firmware logical flow

calling SoftDevice's BLE functions.

I used the SDK V.10 for the nrF51 series and the SoftDevice S110—both supplied by Nordic Semiconductors. Development of firmware/software for BLE peripherals cannot be fully explained in the few pages of an article. So, I'll only summarize the fundamental concepts for understanding this project, referring to the extensive literature that can be found on the Internet.

The basic concepts—also used in the app development—are: GAP, GATT and its objects. GAP (generic access profile) defines the general topology of a BLE network. Connecting devices can have two different roles: central and peripheral. In my project, the central device (acts as a client) is the smartphone, and the peripheral (acts as a server) is the tag with sensors. The peripheral uses GAP during the advertising phase, when they send some frames to be discovered on air from the central device. It is important to note that a central-peripheral device can be connected to multiple devices. In my project, the smartphone can query the entire network of tags one by one, thereby keeping the entire home under control. When a peripheral is connected to a central device, it stops to send advertising data, so another device would not be able to find the peripheral and connect to it.

Every BLE peripheral has a profile GATT (generic attribute profile). It is the top of the ATT (attribute protocol)—a protocol that defines how a server exposes data to a client, and how they are structured. Every profile contains definitions and properties of services and characteristics. When you connect to a device, you use GATT services to communicate. A profile can have one or more services, and each service can have one or more characteristics. Usually services represent features, whereas properties define operations that can be performed on a characteristic, such as read, write, notify and indicate. Every attribute (service and characteristic) is distinguished by its UUID

For detailed article references and additional resources go to:
www.circuitcellar.com/article-materials

RESOURCES

Adafruit | www.adafruit.com

AMS | www.ams.com

Nordic Semiconductor | www.nordicsemi.com

TE Connectivity | www.te.com

Waveshare | www.waveshare.com

(Universal Unique Identifier). The official BLE standard adopted 16-bit UUIDs, while the custom ones get 128-bit UUIDs assigned.

In my project, I define a custom service with two characteristics. One is read only, and contains a string of four values: temperature, humidity, TVOC and eCO_2. The other is writeable and allows commands to be sent to the tag. This second one will be used for features that I would like to develop in the future, such as activating/deactivating a tag from the network, or starting a ventilation system connected to the tag when certain values are reached.

After explaining the basic concepts, it is possible to better understand the firmware logical flow, which is summarized in **Listing 1**. As you can see, after the initialization of the GAP, the advertising phase and the start of the service that exposes the two characteristics, the rest of the firmware is just an infinite loop that reads relative humidity and temperature, writes them to the CCS811 registry and then reads TVOC and eCO_2—updating the values of service characteristics.

THE SOFTWARE

I developed the app using the Android Studio development environment, and an interesting template called Android BluetoothLeGatt Sample that was provided with the environment. This sample demonstrates how to create a custom service for managing connection and data communication with a GATT server. You therefore have every component necessary to transmit arbitrary data between devices by Bluetooth LE API.

Recently, a useful discussion space on GitHub was created about this sample [6]. The GitHub discussion also contains all the modification proposals, as well as code examples. Obviously, I had to change the sample for my purposes. Explaining how to develop an Android app that uses the Bluetooth protocol is beyond the scope of this article, so I will focus on the most significant changes I made.

The sample is created with a default interface. The first step is to design the new interface for viewing the four monitoring parameters. I therefore inserted four `TextView` controls, each connected to a data field. `TextView` is a user interface control that is used to set and display the text to the user. One was for the temperature (`id.temp`), one for the relative humidity (`id.hum`), one for the TVOC (`id.tvoc`), and the last for the eCO_2 (`id.eco2`).

I placed a button linked to the `onClickUpdateData` event, which updates the interface data fields with the values in the service characteristic that correspond to those received by the tag via Bluetooth. The definition of the various interface components is grouped in the iaq_net_layout.xml file. This file is read when the `OnCreate` function (`DeviceControlActivity.java`) is executed during the app boot. The correct layout is loaded thanks to the `setContentView(R.layout.iaq_net_layout)` instruction. In `DeviceControlActivity` I also define the `mTemp`, `mHum`, `mEco2` and `mTVOC` data fields.

To complete the interface development, it is necessary to remove the references to any fields of old layout by inserting those to the new layout fields, such as `mTvoc = (TextView) findViewById(R.id.hum)`, for the relative humidity values. In this way we have linked the interface to the internal variables that will store the measured values.

Figure 6 shows how the app starts by scanning all the Bluetooth devices present in the surrounding area. Tap on the IAQtag link (C8:41:E5:BF:8F:01 MAC address), and the app will connect with the tag displaying the detected data. Now, by clicking on the "Update Data" button, the values are updated in real time.

I then moved on to the development of the functions for processing the values represented in the interface. I added a reading function for the service characteristic `readCustomCharacteristic()` (`BlueToothLeService.java`), and inserted in the function `mGattUpdateReceiver` the

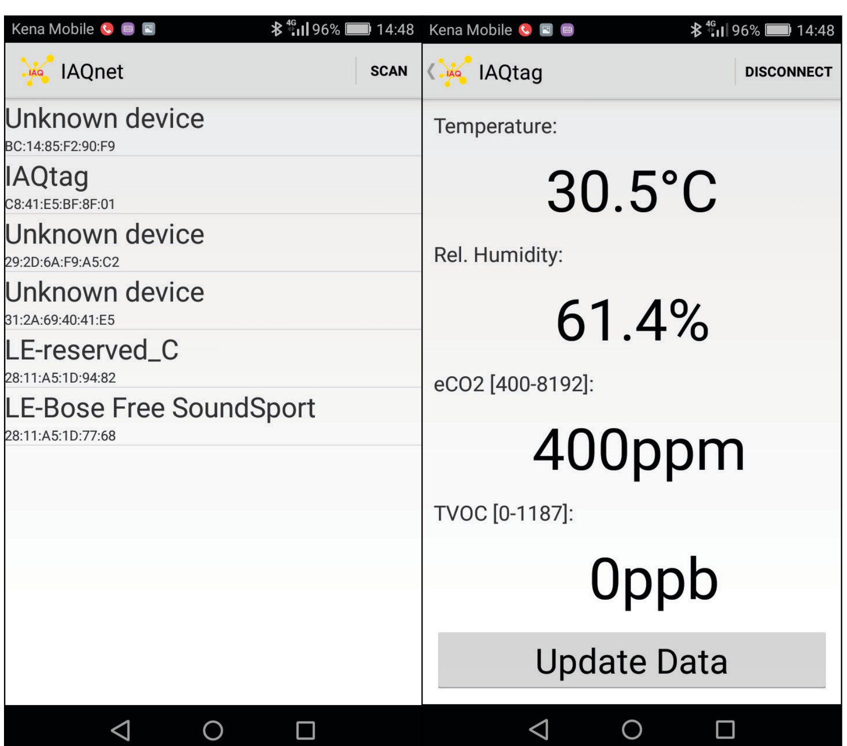


FIGURE 6
Android app IAQnet

FEATURES

```
private void displayData(String data) {
    if (data != null) {
        //Extract data fields from received string
        mTemp.setText(data.substring(0,4));
        mHum.setText(data.substring(4,8));
        mEco2.setText(data.substring(8,12));
        mTvoc.setText(data.substring(12,16));
    }
}

public void onClickUpdateData(View v){
        if(mBluetoothLeService != null) {
          mBluetoothLeService.readCustomCharacteristic();
        }
      }
```

LISTING 2
Data extraction and link to interface button interaction

necessary instructions when the measurement data is ready to be displayed. In the firmware, I chose to send the data from the tag already preformatted in a string 20 characters long. In the reception sequence I call a function to extract the individual values and assign them to the respective internal variables. At this point, all that remains is to link the instructions to the interface button interaction by entering a recall to the `readCustomCharacteristic()` (**Listing 2**).

When you click on the button, you call the function `readCustomCharacteristic`, and the following actions occur: the values are requested from the tag that updates its service characteristic, the characteristic value is read by the smartphone via BLE updating its own service, then the individual values are extracted from the string and inserted into the internal variables. These are linked to the interface that displays values on the smartphone screen.

From Android 5, for all the BLE apps you need to add some permissions in the manifest file to access the device location, and call a function that requires the user to authorize this activity when the app is running. This can be done by adding a `verifyPermissions` in the `OnCreate` event (`DeviceControlActivity.java`), as shown in **Listing 3**.

CONCLUSION

An interesting future development would be to use the notification system to update the data on the Android app. Notifications can be sent to the client periodically or whenever the characteristic value changes. The smartphone can register for these notifications, so ambient parameters (IAQ, Temp, Hum, eCO_2) are automatically displayed, rather than being requested by a refresh command.

This project can serve as the basis for developing any network of environmental tags. It is sufficient to replace the sensors with those needed for your application. The changes to the firmware and the Android app are relatively simple and relate to the same functions implemented in this project. After all, once the tag has taken the measurements and made them available in the service characteristic, they can be requested from the app via Bluetooth.

Another feature I would like to implement is a second service characteristic that is remotely writable. You could use it to encode a series of commands that could be interpreted by the tag, which should execute them as soon as they are sent via Bluetooth. For now, I hope this circuit can be a starting point for those who want to experience the world of Bluetooth and Android development.

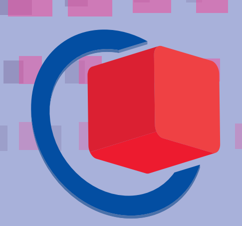
```
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>

public static void verifyPermissions(DeviceScanActivity activity) {
    int permission = ActivityCompat.checkSelfPermission(activity, Manifest.permission.ACCESS_FINE_LOCATION);
    if (permission != PackageManager.PERMISSION_GRANTED) {
        // We don't have permission so prompt the user
        ActivityCompat.requestPermissions(
                activity,
                new String[] {
                        Manifest.permission.ACCESS_COARSE_LOCATION,
                        Manifest.permission.ACCESS_FINE_LOCATION,
                },
                1
        );
    }
}
```

LISTING 3
Permissions and a function to verify them

Nürnberg, Germany

February 25 – 27, 2020



embeddedworld

Exhibition&Conference

... it´s a smarter world

DISCOVER INNOVATIONS

Over 1,000 companies and more than 30,000 visitors from 84 countries – this is where the embedded community comes together.

Don't miss out! Get your free ticket today!

Your e-code for free admission: 2ew20P

Media partners

SmarterWorld

medical design

elektroniknet.de

Exhibition organizer

NürnbergMesse GmbH
T +49 9 11 86 06-49 12
visitorservice@nuernbergmesse.de

Conference organizer

WEKA FACHMEDIEN GmbH
T +49 89 2 55 56-13 49
info@embedded-world.eu

NÜRNBERG MESSE

Sound Localization
Using a PIC32 MCU

By
JinJie Chen and Alvin Pan

Learn how these two Cornell students built a sound localization device. The employed a Microchip PIC32 MCU and a set of microphones to determine the direction from which an arbitrary sound is coming. They recorded input from three microphones to identify the time delay between the audio recordings. These time delays provide a means to calculate the direction of the sound.

It's amazing to see what kinds of sound analysis can be done using a 32-bit MCU. Our project is the construction of a sound localization device. The Microchip PIC32 microcontroller (MCU)-based device is a triangular arrangement of microphones used to localize the direction from which an arbitrary sound is coming. By recording input from three microphones, we were able to identify the time delay between the audio recordings. These time delays provide a means to compute the direction of the sound.

The hardware for the project is made up of three main parts: three microphone circuits, a TFT (thin-film-transistor) LCD (liquid crystal display) and a custom PIC32 prototyping board. The prototyping board gives a breakout for pins on the PIC32 in addition to 3.3V power, an SPI-controlled DAC and an SPI-controlled TFT display. The prototyping board uses the PIC32MX250F128B [1], but theoretically, any PIC32MX MCU should have the same hardware we used.

Each of the three microphone circuits includes an electret microphone, a set of filters and an amplifier. The output of each microphone circuit is fed into an ADC channel on the PIC32. The TFT display is used to show debugging information and to point in the direction of the sound. The full schematic is shown in **Figure 1**.

THREE-PART CIRCUIT

The microphone circuit consists of three parts: The microphone itself [2], a high-pass filter to center the signal around half voltage and an amplifier, which uses an active band-pass filter to amplify only frequencies of interest. A Texas Instruments (TI) LM4562 audio op amp [3] acts as the core component of the amplifier and filter part of the circuit. Because the LM4562 is not a rail-to-rail op amp and does not work with 3.3V, a different set of rails was required to keep the op amp out of saturation. 9V were supplied to the positive rail, and -3V were supplied to the negative rail. Because we found that noise from the MCU can get tracking into the audio circuitry, the 3.3V supply for the microphones was generated from a separate regulator with a constant load constructed out of a few resistors.

The initial high-pass filter has a cut-off of around 160Hz. The high-pass filter on the band-pass amplifier was selected to roughly match the cut-off of the initial high-pass filter. The low-pass filter of the op amp was selected to give roughly a 7.3kHz cutoff frequency. The gain used was 100:1.

Each output from a microphone circuit was attached to an I/O pin with analog functionality. To protect these pins from any over-voltage or under-voltage conditions, a pair of Schottky

diodes and a resistor were added to the output to construct a voltage snubber. The Schottky diodes provide a conducting path in case of one of these conditions, while the resistor limits the current flow through these diodes.

The TFT display shows debugging information and points in the direction of the source of the sound. The part we used is an Adafruit breakout (part number 1480) [4] that provides the TFT display, a TFT display driver and an SD card reader (unused). The code for this was a library that was adapted from the library Adafruit supplied for running the TFT with an Arduino. The TFT breakout uses an SPI channel, along with a few other digital I/O pins. Cornell's ECE4760 course links to a library for the TFT display that was adapted from an Arduino library by Tahmid [5].

A Microchip Technology MCP4822 digital-to-analog converter (DAC) [6] was used for debugging the system but was not used for the project itself. By sending the waveform to the DAC at a rate of 5kHz, we can review what the output of the system looks like. More on that later in the results section.

SOFTWARE AND MATH

First, to locate the direction of the sound, the system needs to record the reading from each microphone channel. Second, the recording of each channel is cross-correlated with the next channel to identify the relative time shift from one recording to the other. Third, the relative timing between each pair of microphone channels is used to compute the direction of the origin of the sound source. The relative direction is computed from the timing differences and the physical arrangement of the microphone placement, to derive the direction of the sound source in degrees. This cycle is repeated as quickly as possible, and the direction estimates are digitally low-passed using an averaging filter to give an estimate of the direction of the sound. Finally, the angle is written to the TFT display to show the result.

The analog outputs from the three microphone channels are connected to the three separate ADC channels. The ADC is configured to operate in a timer-triggered sampling mode, which starts a new sample each time the timer-interrupt flag is raised. It is also set to sample three channels and store the results as 16-bit signed integers in the ADC's internal buffer. The ADC is set to raise an interrupt flag once every three samples. To make the system run at the intended frequency, the timer for triggering the ADC

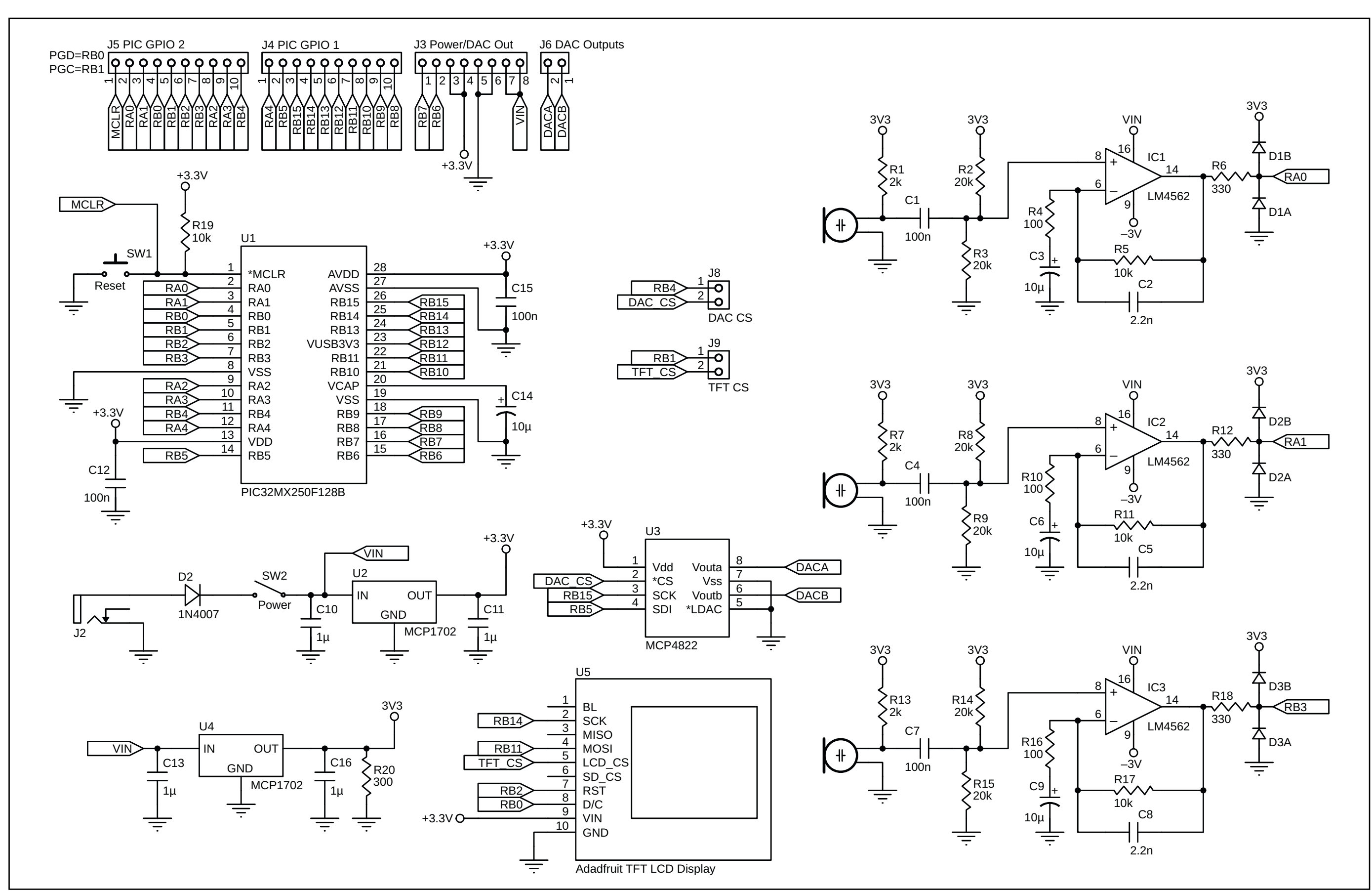


FIGURE 1
Full schematic of the project, including MCU, microphones and power circuitry. Microphone and amplifier circuitry are on the right side.

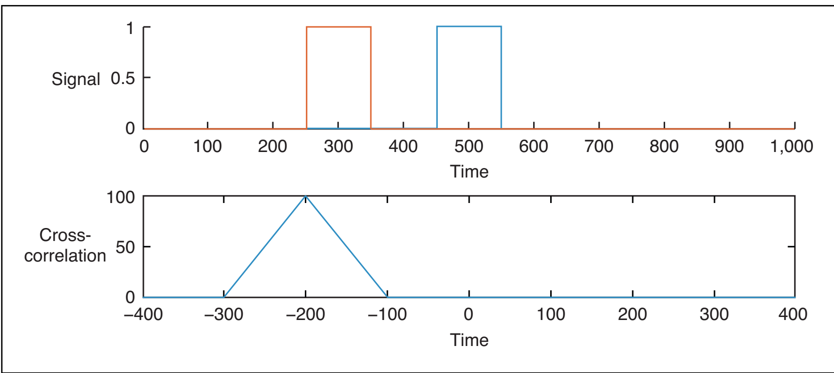


FIGURE 2
Top: two identical square waves with a time shift. *Bottom:* the cross-correlation between the two square waves shown in the top

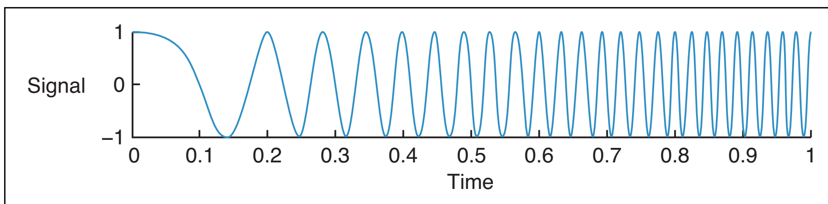


FIGURE 3
An example of a swept sine wave, also referred to as a chirp

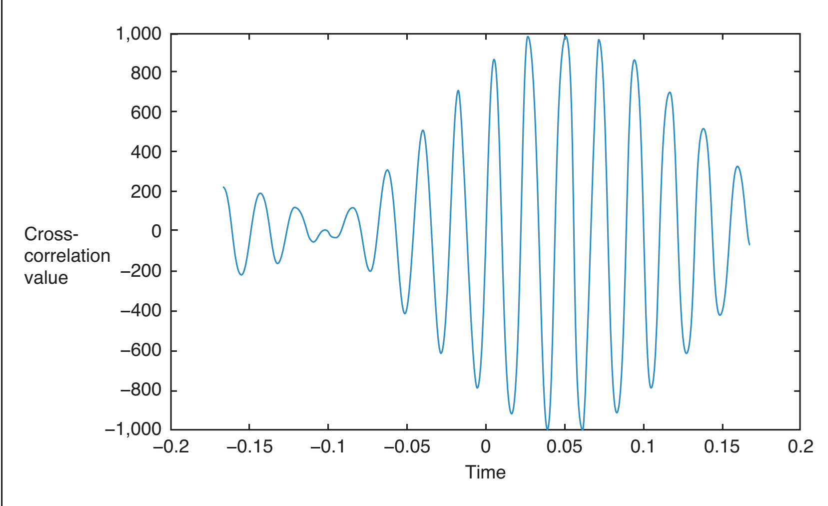


FIGURE 4
An example of the cross-correlation between two swept sine waves with a time shift of 0.05 units

ABOUT THE AUTHORS

Alvin Pan is a recent graduate in Electrical and Computer Engineering and Computer Science from Cornell University with interests in embedded systems and robotics. He can be reached at ap924@cornell.edu

JinJie Chen is a recent ECE graduate of Cornell University and now an embedded software engineer at Google. He is particularly interested in embedded system and edge computing. He can be contacted at jc2554@cornell.edu

sampling is set to run three times faster than the intended sampling frequency.

This causes the timer to trigger three ADC samples—one for each microphone at the intended frequency. A set of three DMA channels is used to transfer the data from the ADC output buffers and into storage. Once the DMA channels are enabled, the DMA transfers 16-bit cells whenever the ADC interrupt flag is raised. When the entire block is transferred, the DMA channel raises the `DMA_EV_DST_FULL` to signal completion of the transfer. The `computation_thread` checks the completion flag for all three DMA channels to determine when to begin the computation of the sound localization.

Once the microphone data is fully recorded in the arrays, as indicated by the `DMA_EV_DIST_FULL` flags, the time delay is calculated between each pair of microphone recordings. The main mathematical technique we use to compute the time delay between two signals—the microphone recordings—is cross-correlation, which measures the similarity of two signals by taking the sum each pair of points in the signal as one signal slides along another. The formula is as follows:

$$(f \times g)[n] = \sum_{m=-\infty}^{\infty} f \times [m] g[m+n] \quad [1]$$

Each correlation value gives the similarity value between the first signal and the second signal, shifted by some amount of time. The index of the entry for the maximum value is the time delay in units of the sampling rate. For example, for the two square waves shown in **Figure 2**, we see the result of taking the cross-correlation (bottom graph).

CROSS CORRELATION

The cross-correlation gives a peak at the maximum overlap. In this case, the orange curve was used as the signal *f* in the equation, and the blue curve was used as the signal *g*. The maximum overlap is the time shift of *f* with respect to *g*, which in this case is at -200 in both plots. Although we referred to the position of the peak as an index, we don't mean an index into an array. It simply means the location in the signal. The index of -200 means that the blue curve must be shifted backward in time by 200 time units to match the orange curve. To get a long signal that we can easily measure, we used a swept sine wave such as the one shown in **Figure 3**, since a swept sine wave is not repetitive. This prevented a situation where the correlation gives a match for multiple different time shifts.

Figure 4 shows the result of taking the cross-correlation of a swept sine wave with another swept sine wave, in which the distinct

```
// cross-correlate each pair of microphone recordings
void cross_correlate() {
  int channel, idx, shift;
  for (channel = 0; channel < num_mic_channel; channel++) {
  // shift the kernel -max_diff to max_diff of mic_data_size
  int correlate_channel = (channel + 1 > 2) ? 0 : channel + 1;
  for (shift = -(max_diff); shift <= (max_diff); shift++) {
    long tmp_sum = 0;
    // kernel size is mic_data_size - (2*(max_diff) + 1)
    for (idx = (max_diff) + 1; idx < (mic_data_size) - (max_diff); idx++) {
      int idx2 = (idx + shift);
      tmp_sum += (((long) mic_data[channel][idx] + mic_bias[channel]) * ((long) mic_
       data[correlate_channel][idx2] + mic_bias[correlate_channel]))>>2;
    }
    // save correlation results for DAC debugging
    correlate_data[channel][shift + (max_diff)] = (tmp_sum > 0 ? tmp_sum : 0)>>3;
    // update new max peak
    if (correlate_data[channel][shift + (max_diff)] > peak_max[channel]) {
      peak_max[channel] = correlate_data[channel][shift + (max_diff)];
      peak_index[channel] = shift + correlate_bias_adj[channel];
    }
  }
  if (abs(peak_index[channel]) < 2)
    peak_index[channel] = 0;
  }
}
```

LISTING 1
The code for computing a cross-correlation via direct application of the definition. Only the center region of on signal is used to avoid the need to normalize the results.

FEATURES

maximum gives the relative time shift between the signals. In this case, the maximum is at 0.05, which indicates that the second signal is shifted 0.05 time units ahead of the other.

In the process of computing the true direction, we evaluate a few constants. First is `mic_data_size` which is the size of the array that holds the microphone data. This is computed by taking *[sampling rate]* × *[sample duration]*. Another is `max_diff` which we use as the maximum value of the time shift of the cross-correlation peak. `max_diff` is the max possible shift geometrically possible, and is computed by:

$$[\text{length}] \times \frac{[\text{sampling rate}]}{[\text{speed}]} \qquad [2]$$

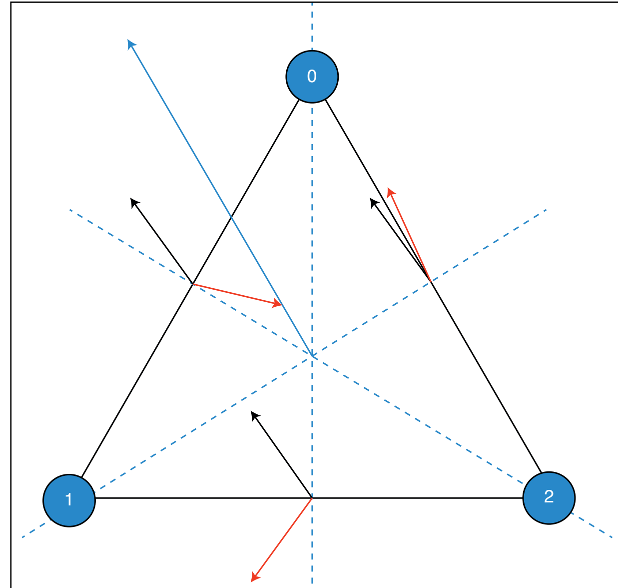


FIGURE 5
An example in which the sound comes from the direction indicated by the large blue arrow. The other smaller angles show the angle estimates given by the correlation between the two microphones. The tails of the arrows start from the edge that sits between the two microphones that were correlated together.

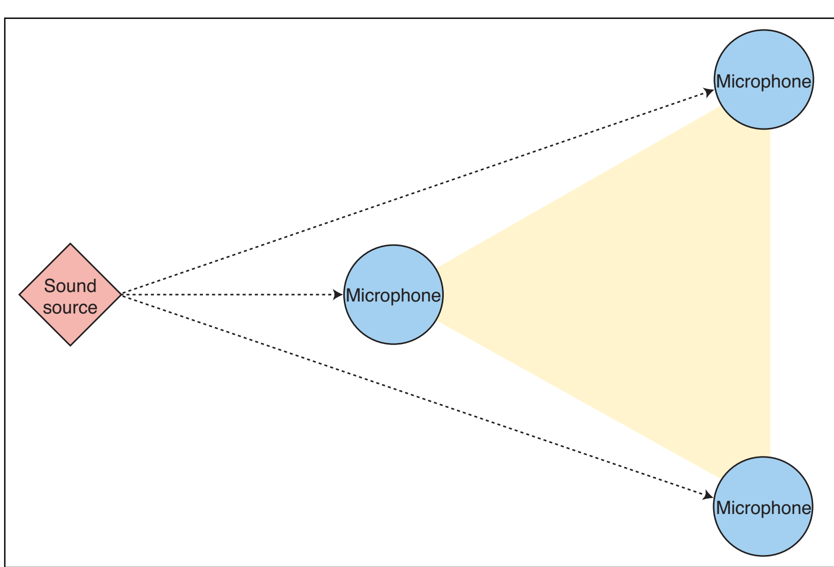


FIGURE 6
The arrangement of the microphones with an example sound source. Computation for the sound directions assumes that the sound source is far from the microphones.

where length is the length of one leg of the triangular arrangement and speed is the speed of sound. These values set how much data is sampled and what region of the sampled data is used in cross-correlation, as discussed next.

The cross-correlations are calculated on channel 0 with respect to channel 1, channel 1 with respect to channel 2, and channel 2 with respect to channel 0. Each cross-correlation is computed by sliding the middle (*mic_data_size - (2 × max_diff)*) wide section of the first recording fully along the second recording, to compute the sum of dot products of the fully overlapped recordings. The resulting cross-correlation values are stored in an array of size *(2 × max_diff) + 1*. Care must be taken to ensure that *(2 × max_diff) + 1* is reasonably smaller than *mic_data_size* to ensure a sufficient number of data points are used in the computation. As the cross-correlation values are computed, the peak value and its associated time shift of each of the three pairs are identified and recorded to compute the direction of the source sound.

Listing 1 is the function used for the cross-correlation. In this code, we compute cross-correlation using the definition equation. A section of one of the inputs is shifted across the other input. Solving the cross-correlation via an FFT (fast Fourier transform), element-wise multiplication and IFFT (inverse FFT) has a better run-time complexity for very large inputs. However, we did not pursue this in our project, because the coefficient of the run time is unknown, and our input size is bounded by the parameters we define.

The direction of the sound source is computed using the three `peak_index` values identified in the cross-correlation calculations. In this case, the name "`peak_index`" is a bit of a misnomer, since these values actually represent the time shift, and not an index into an array.

To measure the direction, the time delay between each pair of microphones is used to compute an angle for each pair. The angle is computed by using the distance between the two microphones and the distance sound travels in the measured time delay. That is:

$$\text{angle} = \arccos\left(\frac{[\text{measured time delay}] \times 343[\text{m/s}]}{[\text{mic distance}]}\right) \qquad [3]$$

This angle is calculated between two microphones, but leaves ambiguity for which side of the two microphones the sound comes from. Each time delay results in both the black arrow and the red arrow (**Figure 5**). We compute an angle for each pair of microphones and then use the arrangement

of the microphones, shown in **Figure 6** to remove the ambiguity.

Using each time shift, we can determine to which microphone the correlation indicates that the sound is closer. For example, in Figure 5, each pair of arrows sitting between a pair of microphones points toward one of the two microphones. In the case of Figure 5, these indicate microphone 0 for each of the upper two legs of the triangle, and microphone 1 for the lower leg. Using the arrangement of the three microphones, as long as the time shifts do not all indicate different microphones, a 60-degree range can be selected, as depicted in Figure 5 by the dotted dividing lines.

ANGLE ANALYSIS

As noted earlier, each angle estimate gives two possible angles. In Figure 5, the correct angle is marked in black, and the false angle is marked in red. For any of the 60-degree regions, one angle estimate always faces outward (of the triangular arrangement), and one faces inward. The remaining angle estimate is ambiguous. The outward-facing angle is the angle computed from 0-1, the inward-facing angle is the angle computed from 1-2 and the ambiguous angle is the angle computed from 2-0. Note that if the direction the sound came from was a bit closer to microphone 0, then the correct direction would be outward-facing rather than inward-facing, which it is now. First, the two angle estimates that are on known sides are computed and averaged together. Using this averaged value, the side for the ambiguous side is chosen by evaluating which is closest to the averaged value. This gives the last angle estimate. All three of these values are then averaged together, which gives the final angle estimate.

One of the six possible ranges of 60 degrees is selected by taking the sign of the time shift and converting it into a binary encoding. This

```
int val = (peak_index[0] > 0) << 2 | (peak_index[1] > 0) << 1 | (peak_index[2] > 0);
int idxP = -1, idxN = -1, idxU = -1;
switch (val) {
  case 0b110: //0-60
    idxP = 0;
    idxN = 1;
    idxU = 2;
    break;
  case 0b010: //60-120
    idxP = 0;
    idxN = 2;
    idxU = 1;
    break;

    ...

}
int x;
for (x = 0; x < 3; x++){
  lim_index(peak_index[x]);
  angles[x] = ((double) peak_index[x])/((double)(max_diff));
  angles[x] = acos(angles[x]);
}
angles[idxP] += angles_adj[idxP];
angles[idxN] *= -1.0;
angles[idxN] += angles_adj[idxN];
lim_angles(angles[idxP]);
lim_angles(angles[idxN]);
angle = angles[idxP] + angles[idxN];
if (range_checker(idxU, angle))
  angles[idxU] *= -1.0;
angles[idxU] += angles_adj[idxU];
lim_angles(angles[idxU]);
double x_pos = cos(angles[idxP]) + cos(angles[idxN]) + cos(angles[idxU]);
double y_pos = sin(angles[idxP]) + sin(angles[idxN]) + sin(angles[idxU]);
angle = atan2(y_pos,x_pos);
```

LISTING 2
The code for computing the sound source's direction. The [...] section omits additional cases for brevity.

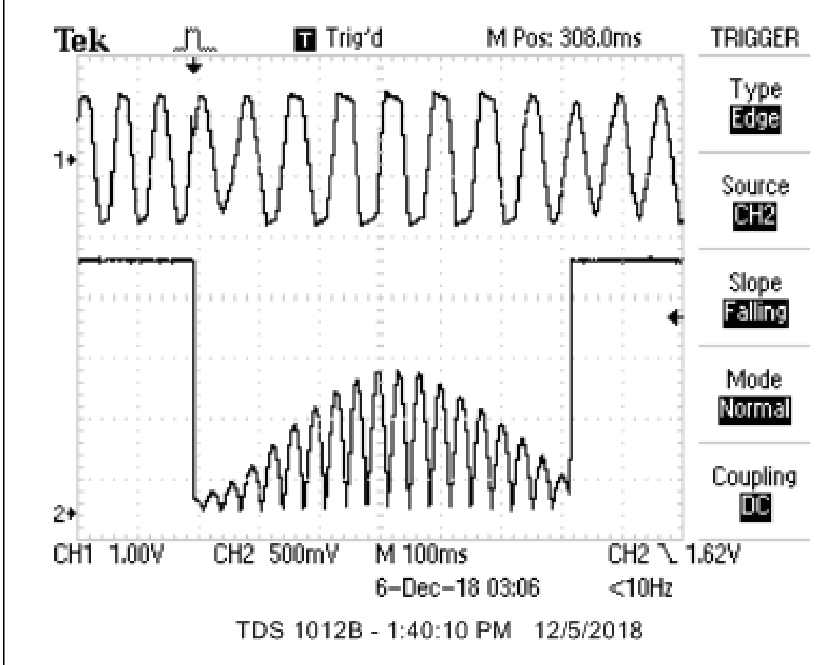


FIGURE 7
Oscilloscope traces of the original waveform of the cross-correlation computation output from an early prototype. The upper trace illustrates recorded signal from one of the microphones, and the bottom trace shows the cross-correlation result from a pair of the microphone channels. Traces such as shown were used for debugging the system.

is done in the `switch` statement in **Listing 2**. Only the first two cases were included in **Listing 2** for brevity. It turns out the computations for each of the six ranges are similar, and we simply need to swap which angles estimates are the outward, inward, and ambiguous ones. This is done by using `idxP`, `idxN` and `idxU`, which represent the index of the outward angle, the index of the inward angle and the index of the ambiguous angle, respectively, where index here identifies which cross-correlation time shift is used. The helper function `range_checker` determines if the ambiguous angle is outward or inward, based on the angle estimate given. The array `angles_adj` holds the offset of each angle estimate with the baseline direction. In this project, it is the direction of microphone 0. The other helpers `lim_angles` and `lim_index` limit the range of values to within the range of angles and indices, respectively.

The low-pass filter, which averages the new angle with the old angle to produce the result, is not shown. The process is the same as averaging the three angles, except that the component values of each vector are weighted to set the cut-off frequency of the low pass.

Averaging angles has a pitfall, in that angles wrap around. Say we wish to average two angles, 170 degrees and -170 degrees. We would like this to give the value 180 or -180 degrees, but a simple averaging of the angles gives the angle 0, which is the exact opposite of what is desired. To average the angle correctly, we instead convert each angle into a unit vector and average the components. The average vector is then converted back into an angle, giving the average angle.

For detailed article references and additional resources go to:
www.circuitcellar.com/article-materials
References [1] through [7] as marked in the article can be found there

RESOURCES

Microchip Technology | www.microchip.com

Texas Instruments | www.ti.com

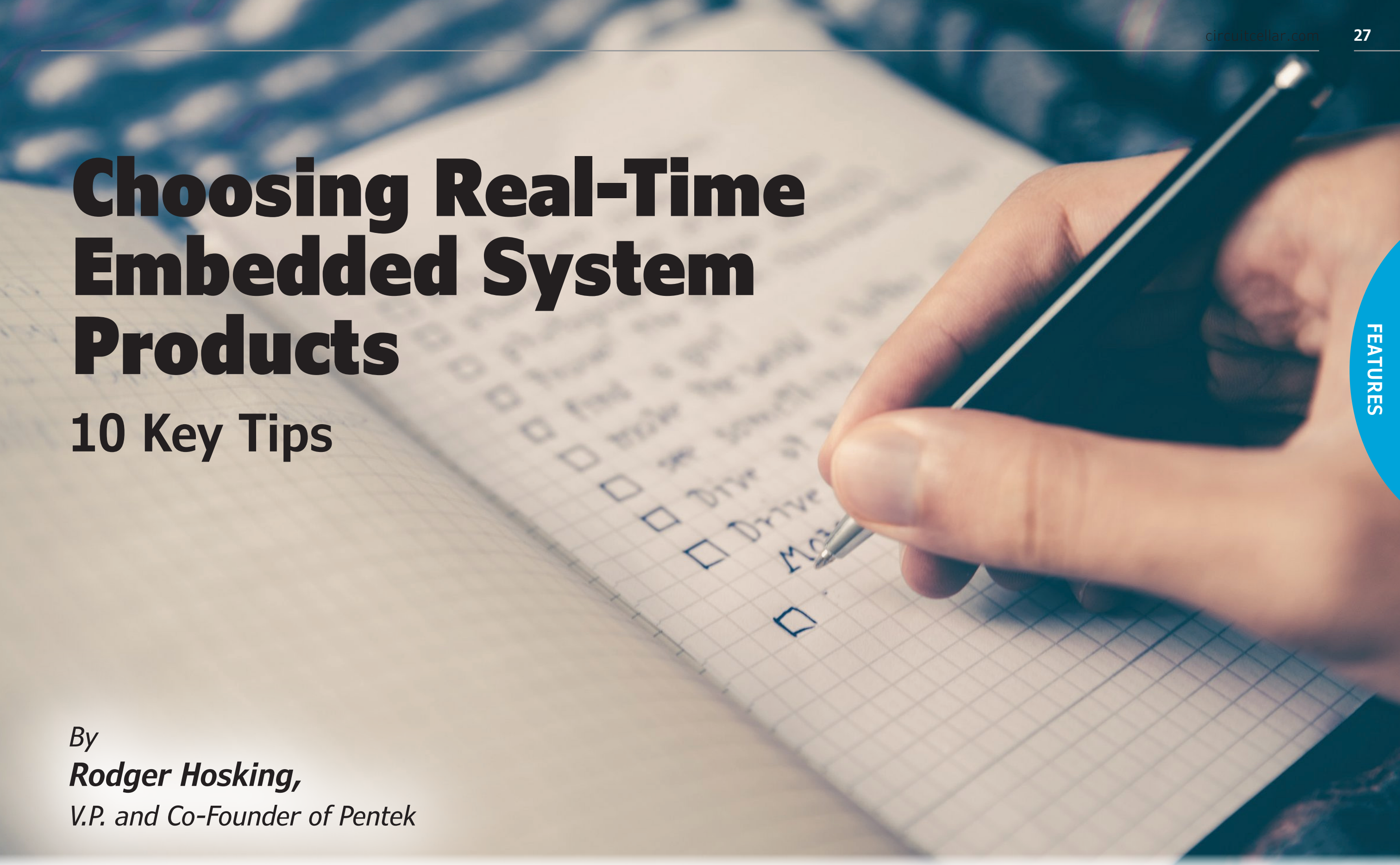
RESULTS

The sound localization worked reasonably well. Although the computation delay was almost indistinguishable when we ran single sweeps, the delay turned out to limit the max accuracy of the system, since it relies on an average of multiple sweeps. The final version of the device used 20cm legs on the triangular arrangement, an 80kHz sampling rate, and 0.025 second sampling time. The remaining parameters were all derived from these values and computed at compile time, using C macros. Using multiple sweeps, we were able to get the system to home in on the direction of the sound. **Figure 7** shows oscilloscope traces of the original waveform of the cross-correlation computation output from an early prototype.

The upper trace in Figure 7 illustrates recorded signal from one of the microphones, and the bottom trace shows the cross-correlation result from a pair of the microphone channels. A high signal is used on the cross-correlation trace to show when the data start and end. This gives us a point at which to set the oscilloscope to trigger, and allows us to see where the start and end of the cross-correlation are, along with the relative location of the peak. In this image, the peak is roughly centered, showing that the sound signals arrived at the two microphones at the same time.

The plot differs from the cross-correlation of the swept sine wave examined in the math background, because this plot takes the absolute value of the cross-correlation. In testing, we found that swept sine waves picked up by the system almost always had nicely formed cross-correlation plots, such as the one shown in Figure 7. However, we found that the location of the peak wouldn't always be in the same place. Further testing and experimentation showed that adjusting the circuit to have a well-defined phase shift for each frequency was key to making the system work. It's essential that all filtering and amplification circuits have the same phase shift for every frequency.

After reworking the circuitry and moving to op amps with a higher gain-bandwidth product (from the MCP6242 to the LM4562), we found that the system appeared to get the angle correct to within 30 degrees in the worst case, and usually within 15 degrees of the correct location. In a more controlled environment, it is highly likely that the system would achieve better performance. The environment in which we tested the device was cluttered with lab equipment, which gave reflected sound waves and multipath distortion. Our code is based on examples given in Cornell's ECE4760 course website [7]. The linked pages also contain the example code that our code uses as a basis.



Choosing Real-Time Embedded System Products

10 Key Tips

By
Rodger Hosking,
V.P. and Co-Founder of Pentek

There are many factors to consider when selecting components and board-level solutions for a real-time embedded system. In this article, Pentek's Rodger Hosking steps through 10 key tips that can help you significantly avoid risks and reduce development efforts.

Real-time embedded systems require a specialized class of electronic components from vendors that can support the special needs of systems integrators. Not only must the hardware products operate across a wide range of operating modes and environments, they must also deliver performance levels meeting critical objectives for specific applications. But other factors may be even more important.

Because each project is unique, good system development tools are essential for systems integrators to deliver operational systems to their final customers as efficiently and effectively as possible. And, the vendors of these hardware and software products must help integrators choose the most appropriate products, support them during development, and then offer life cycle management solutions for continued product availability.

By following some key recommendations in making product and vendor choices, integrators can significantly avoid risks and reduce development efforts. A summary list of these tips is shown in **Table 1**.

HARDWARE TIPS

Open Standards: Increasingly, both government and non-government procurement requirements now mandate or encourage compliance with emerging open-system standards for embedded hardware components. Among the many benefits are interoperability among vendors, faster deliveries and competitive pricing. Instead of replacing an entire system for a new technology upgrade, open standards allow replacement of compliant modules more quickly and at far less cost, thus extending the useful life cycle of deployed systems.

By following open standards, vendors also benefit by focusing design and development efforts on their areas of expertise, while other vendors produce complementary and compatible products to round out the supply chain. This fosters government confidence in relying upon these open standards for future long-term programs.

Thermal management: As silicon device geometries continue to shrink, the power dissipation per transistor or element tends to drop, but this is often offset by increased

Tips for Choosing Real-Time Embedded Systems
Choose open standard products for best value and life cycle
Define thermal management strategies early in the project
Identify all interconnections, including speeds and levels
If required, define early on how channels are synchronized
Identify all necessary clocking, timing, and DMA functions
Check for software, drivers, and examples of the above
Look for high-level C libraries with underlying source code
Identify which FPGA structures are included and supported
Decide who performs the required custom FPGA design efforts
Ensure graphical FPGA design entry tools support your boards
Look for AXI4-complaint FPGA IP blocks from the board vendor
Look for FPGA application examples from the board vendor
Understand the board vendor's applications support policy
Look for the board vendor's life cycle management programs

TABLE 1
Checklist of critical tips for choosing real-time embedded system products

clock rates. In addition, more elements can fit in a given size package, which drives power per device back up. Fine-pitch ball grid array packaging boosts component density of PCBs, often causing significant heat per slot in today's embedded systems.

When designing a new embedded system, it is imperative that designers identify heat sources for each module as early as possible. For harsh environments, the popular VPX specification defines numerous available solutions, including forced-air, air flow-though, conduction-cooling and liquid cooling. Less demanding environments are often suitable for PC platforms. Selecting the most appropriate platform and thermal management strategies at the beginning of a project can avoid costly redesign cycles and serious delays.

System Interfaces: One of the toughest challenges in real-time system design is connecting system elements via data interfaces capable of handling the required traffic. Start by creating an overall system block diagram containing all essential elements, showing the data interconnect paths between them. Make sure that the interfaces on connected blocks match in type, bus width, lanes and data rates, and enter that information on the block diagram. Then, calculate and notate the worst-case data transfer rate required for each path, and compare it with the maximum path rate. This assessment of interconnect speeds will be an invaluable reference during development.

Some caveats to watch out for are interfaces that pass through several physical connectors, like a PCI Express link from an XMC module, through an XMC carrier that is plugged into a VPX backplane. Every connector can compromise maximum achievable rates. In these cases, modeling or functional test verification can help.

Shared resources, such as system memory, may have multiple contenders for access, resulting in compromised availability. Signals like received radar pulses can generate blocks of high peak rates separated intervals with no data. An elastic memory buffer (FIFO) may be required to take advantage of the low average rate for transfer across the interconnect path.

Synchronization: A growing number of phased-array antenna application, including 5G wireless, airborne and SAR radars, and directional communication links, all require multiple element antennas to support beamforming for receive and transmit (**Figure 1**). Each antenna element signal requires precisely controlled, programmable phase shifts relative to all of the other elements.

Each signal often connects to a dedicated data converter where DSP circuitry can easily handle these precise phase shifts. However, the data converters must acquire and generate each sample at exactly the same sample clock edge. For large arrays, the high number of elements may require synchronous operation across multiple boards or chassis to handle all the channels.

Such operation can only be achieved if this feature is part of the board design, and supported with timing and sync generators connected to each board. If channel synchronization is part of the system requirement, make sure the boards inherently include this feature with recommended

FIGURE 1
Massive MIMO (phased array) 5G Wireless antennas can enhance signal coverage at specific elevation and azimuth angles for faster speeds and more users.

connectors, cables and sync generators, because synchronization is otherwise nearly impossible to add later.

Installed Features: Real-time embedded boards playing typical roles in any system should include several basic functions supporting their assigned roles. For example, a single board computer (SBC) or PC motherboard will almost always implement a PCI Express root complex, system memory mapped across PCI Express address space, network interfaces, and CPU peripheral I/O like USB, serial and video. Standard CPU chip sets include virtually all of these functions, and supporting software drivers for Windows or Linux are commonly provided by the board vendor.

Other boards, like FPGA software radio modules with A/D and D/A converters, have far different roles and requirements. Commonly needed functions here include triggering, gating, time-stamping and synchronization engines that meet tight timing demands. Sample clock frequency synthesizers should accept a 10MHz system reference from an onboard GPS receiver or external source. DMA controllers must move data between the data converters and system memory through a PCI Express interface. Memory controllers for external SDRAM must buffer and capture real-time data converter streams, and communicate with the PCIe interface.

Unlike SBCs or PCs, which benefit from standard chip sets with low-level BIOS initialization, none of this exists for software radio boards. Instead, each of the hardware resources must be developed and incorporated in the FPGA. Equally important are the software libraries and drivers needed to make all of these resources work as required. Unless the board vendor includes them as factory installed features along with the supporting software libraries, the system integrator must develop, design, test, and document this on his own. To minimize risks, expense, and uncertain delays, systems integrators should make sure the board vendor includes these important resources.

DEVELOPMENT TIPS

Software Development: Although open-system architectures help with electrical and mechanical interoperability, all real-time embedded systems are a collection of diverse hardware elements that must be carefully configured for a specific, unique application. Unlike mass market PC boards with plug-and-play capabilities, most embedded boards must be explicitly configured to perform specific tasks, told how to utilize specific external input, output and timing signals and instructed what, when and how to communicate with other boards in the system. This is invariably accomplished by writing custom C programs that execute on the system controller, typically running Linux or Windows OS.

Even if an embedded board vendor provides C-callable functions for programmable hardware features, those offerings vary widely among vendors in their completeness and usability. Some offerings simply provide access to the programmable registers for the devices on the board, and the developer must use data sheets from the device manufacturer to figure out which bits to set. Even with a detailed block diagram of the board, this is very cumbersome.

In a far better approach, the board vendor offers high-level C libraries with well-documented command parameters that relate to the overall board-level operations performed, including references to other operations affected. Each of these high-level commands should include a well-organized underlying collection of low-level libraries to allow modification for specialized operations.

An even more elegant offering is a true API (application programming interface) with an API command processor program running on the system controller. In this way, API commands can be sent to the controller where they are parsed and executed, without needing to recompile a dedicated, executable C-program. API commands can be delivered to the controller via Ethernet, nicely supporting control and status functions of the embedded system from a remote client.

Last, numerous C program examples that illustrate typical operating scenarios are extremely valuable. They incorporate multiple high-level function calls with comments explaining the purpose of each, including why they must be executed in a specific order. Often these fully-tested examples can be incorporated directly into the final application to speed development. Of course, full C source

ABOUT THE AUTHOR

Rodger H. Hosking is vice-president and co-founder of Pentek, where he is responsible for new product definition, technology development and strategic alliances. With over 30 years in the electronics industry, he has authored hundreds of articles about software radio and digital signal processing. Prior to his current position, he served as engineering manager at Wavetek/Rockland, and holds patents in frequency synthesis and spectrum analysis techniques. He holds a BS degree in Physics from Allegheny College and BSEE and MSEE degrees from Columbia University in New York.

FEATURES

code should accompany all library functions and code examples.

By selecting vendors offering these higher-level tools, systems integrators can complete their development tasks much more quickly and will be able to support changes and future upgrades far more easily.

FPGA Development: FPGA designs are really hardware designs, in which the basic hardware resources of the FPGA (thousands of gates, adders, multipliers, registers, switches, memories and interfaces) are wired together to create custom circuits. The wiring connection pattern is generated by software tools from the FPGA vendor that compile descriptive instructions from the designer to create a "bitstream." When loaded into the FPGA, the bitstream implements the required interconnects for the required circuit, often simply called "IP."

As mentioned earlier, the board vendor may install some standard IP functions at the factory. But many customers need to install additional custom IP within the FPGA for compute-intensive, real-time algorithms. These algorithms are often the systems integrators' "secret sauce," comprising their critical value-added contribution to the equipment. The ease of adding IP by the customer is highly dependent on the quality of the FPGA design package supplied by the board vendor.

FIGURE 2
Pentek Model 5950 Zynq UltraScale+ RFSoC 8 Channel A/D and D/A VPX module.

First of all, look for a board vendor that includes most of the essential factory-installed features, like the ones described earlier. It will dramatically reduce the overall FPGA design effort. No one wants to spend years developing a JESD204 data converter interface or a DDR4 SDRAM controller!

Next, be sure the board vendor supplies FPGA source code for all of the installed IP modules in the HDL format matching your FPGA designers' capabilities, usually VHDL or Verilog.

Ideally, all IP from the board vendor will be delivered as AXI4 compliant blocks to match the style of reference IP blocks from the FPGA vendor. AXI4 is a widely adopted interface standard derived from ARM technology that tackles most of the housekeeping chores for connecting one IP block to another.

Take full advantage of the graphical design entry tools from the FPGA vendor, such as Xilinx's Vivado IP Integrator. All of the AXI4 blocks are visually displayed, representing the entire block diagram of the project with all interconnects shown. IP blocks and interconnects can be added, deleted, and modified with mouse clicks, and hyperlinked documentation is available by clicking on any block. After making the required changes, the project is recompiled to produce the new design and bitstream for the FPGA.

Choose a board vendor that delivers the entire FPGA project folder containing all the files needed to create the delivered FPGA IP, fully AXI4 compliant, with complete documentation, and ready to compile using the FPGA vendor's tool suite.

VENDOR TIPS

New Technology: One of the major benefits of open standard COTS products is upgradability of existing systems with new technology by replacing a module instead of scrapping the system and starting over. Of course, depending on the upgrade, changes will often be needed to system software and perhaps even to some of the other hardware, interfaces or connectors. Still, this is a well-proven strategy for extending the useful life of deployed equipment.

Choose board vendors with a history of consistently delivering open-standards products based on each new generation of FPGAs, data converters, memories, and system interfaces. Look for high-level development tools from those vendors to simplify the migration of software and FPGA designs.

For detailed article references and additional resources go to: www.circuitcellar.com/article-materials

RESOURCES

Pentek | www.pentek.com

Applications Support: Because every embedded system tends to be unique, systems integrators invariably encounter first-time configurations of multi-vendor products that don't seem to work as expected. Too often, each vendor blames another vendor for the problem, leaving the integrator on his own. Choose vendors with a proven track record of solving problems, regardless of who is at fault and share such experiences with other project teams.

Most board vendors offer contracts to provide technical support during the development phase, although the quality and timeliness of that support varies among vendors. When the support contract runs out, before they can get additional help, customers will either be asked to renew the contract, or for a credit card number. Some vendors offer free support for a limited time or number of hours, with payment required thereafter.

Be sure to ask any potential vendor for written descriptions of the applications support policies and costs before purchasing his products.

Life Cycle Management: Often, a significant concern for systems integrators is the increasing prevalence of component obsolescence, or end-of-life. This causes two major problems. Future component availability can jeopardize on-going production of enough boards to support multi-year installation program cycles. Also, 20- to 30-year maintenance contracts to support these fielded systems are at risk without components needed for repairs.

Systems integrators naturally look to the board vendors for help, and various strategies have emerged. The simplest one is to purchase and produce enough additional boards up front to cover all installations over the life of the program, plus spares to cover the expected number of failures. End customers usually balk at the cost of this approach.

A very cost-effective alternative is a bonded inventory component program. The board vendor purchases all of the active components needed for the production of the total number of boards required over the life of the program, plus extras for repairs. The customer agrees to pay for these components, which the vendor reserves for him in bonded inventory. When production is required, those parts are used and their cost is credited toward the new purchase.

Since components such as PCBs and hardware can always be purchased as needed for later production, the cost of this bonded inventory program is a small fraction of the cost of full production up front, and very attractive to most customers.

PUTTING IT ALL TOGETHER

As an example of these strategies developed over three decades, Pentek's latest offering is the Model 5950 Quartz RFSoC 3U VPX module (**Figure 2**). Following the VITA 65 OpenVPX standard, this powerful software radio board combines eight channels of wideband A/D and D/A conversion, a wealth of Xilinx Zynq UltraScale+ FPGA resources, and a multi-core Arm processor to handle system controller functions.

Factory-installed features include IP for wideband data acquisition, triggering, timing, and multi-channel synchronization. A waveform generation engine creates analog signals from customer-created waveform tables or from an on-board frequency synthesizer and chirp generator. Linked-list DMA controllers move data from the board to and from the PCIe Gen.3 x8 interfaces and two 100GigE interfaces, each capable of sustaining 12GB/s.

All of these resources are supported with software development tools under the Pentek Navigator Board Support package. It includes a high-level API, C-language libraries, a command processor for the Arm, complete C source code, and fully functional starter applications. For custom FPGA development Pentek's Navigator FPGA Design Kit contains the complete Xilinx Vivado project for the Model 5950, and a library of over 140 Pentek AXI4 IP modules for adding new features.

Pentek offers free lifetime applications support and well-development life cycle management and bonded inventory programs. By introducing a constant stream of industry-leading, open standard board level products with the latest data converters and FPGAs, Pentek helps systems integrators to take earliest advantage of the newest technology.

LCR-Reader
Digital Multimeter

LCR-Reader-MPA
Ultimate PCB debugging tool

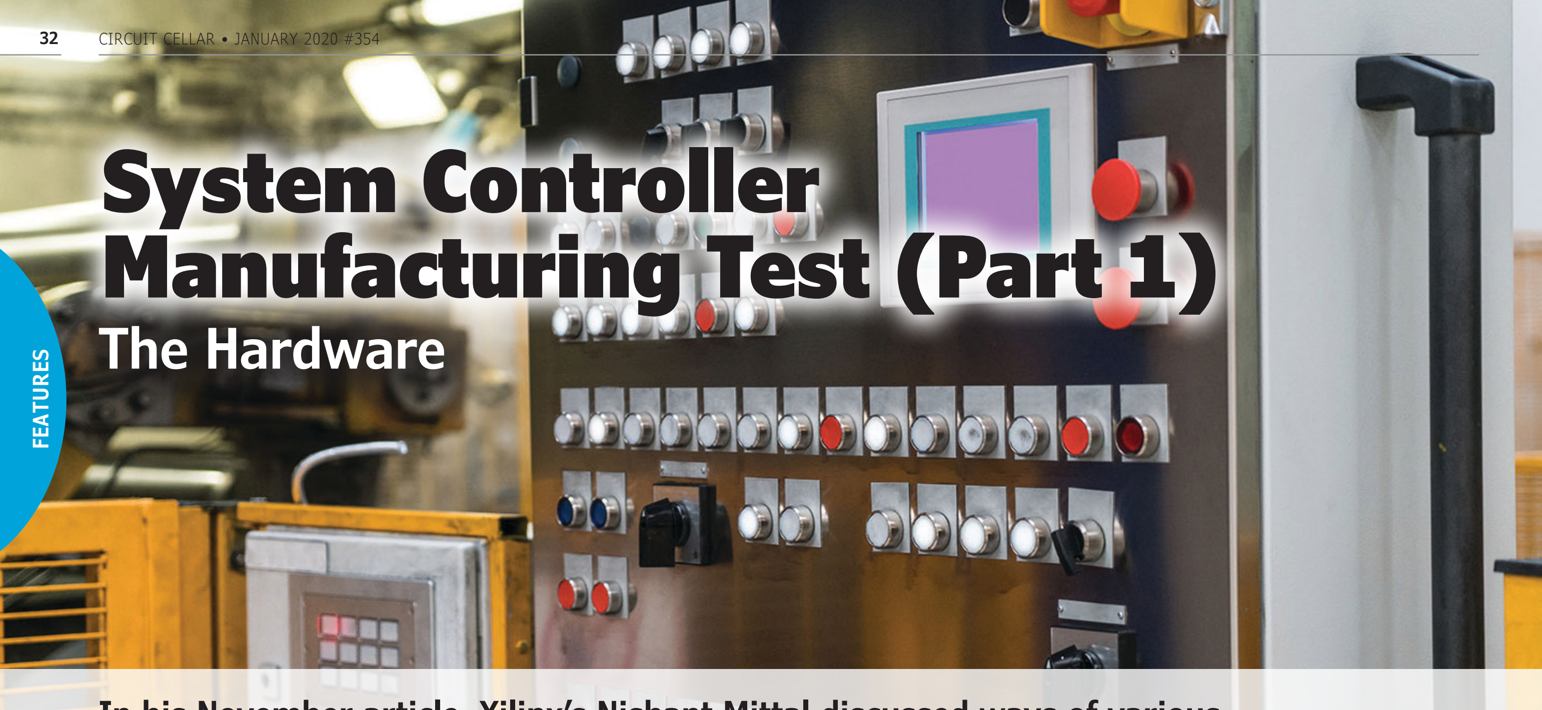
L-C-R, AC/DC Voltage/Current
LED/Diode/Continuity Test
Oscilloscope
Frequency, Period, Duty Cycle
Signal Generator
Super Cap Testing

Basic Accuracy 0.1%
Test Frequency: 100 Hz to 100 kHz
Test Signal Level: 0.1, 0.5, 1.0 Vrms

All-in-One
Digital Multimeter

LCR-Reader.com

FEATURES



System Controller Manufacturing Test (Part 1)

The Hardware

In his November article, Xilinx's Nishant Mittal discussed ways of various ways of testing a board. In this two-part series, Nishant expands on that topic, this time discussing the design of an FPGA-based system controller built for testing and managing complex platforms. Part 1 focuses on the hardware aspect of the system, including the hardware design, building blocks, algorithm and so on.

By
Nishant Mittal and Manoj Khandelwal

A "system controller" can be defined as a system on a board or a platform capable of managing, controlling and monitoring the entire platform—right from power to communication. A system controller can be used not only to manage a platform, but also to test the peripherals of the platform, which reduces the cost of manufacturing test.

As electronic devices become more complex, the platforms for these devices also have become huge. Test coverage of the entire board for various features can become difficult especially when it's a SoC that has multiple peripherals with different power controls on it. Such types of systems require an equally competent controller onboard that can easily manage and control the entire set of knobs. In my article "Designing Manufacturing Test Systems" (*Circuit Cellar* 352, November 2019) I discussed, various ways of testing a board. In this article, we will discuss one aspect of that with a system controller on board. All that said, much of this article will focus on the management of knobs and monitoring the system.

After the initial development, this can act as a "black box" and can sit on any other platform to perform different actions. This is presented in two parts to help readers understand how to design a system controller using Xilinx Zynq Ultrascale+ FPGAs and Xilinx tool chains. Here, we'll discuss the hardware aspects of the system including the design, the building blocks, the algorithm and so on. In Part 2, coming next month, I'll discuss the software and firmware of the project as a part of complete system integration.

DESIGN STEPS

Designing a system controller involves brainstorming from both a hardware and a software point of view. **Figure 1** shows the block diagram of a generic system controller. A typical system controller needs a robust processor, a communication block, a memory block, a clock and a power management block. Any number of additional features could be added to this list.

As shown in Figure 1, we have used the Zynq Ultrascale+ FPGA as the central core of the system controller. The Zynq Ultrascale+ device is broadly divided into two parts: PL and PS. The PS part is the Arm processor while the PL part is Xilinx proprietary hardware block, which does actual FPGA related tasks. To understand details on how Zynq Ultrascale Plus device works, you can read the technical manual [1].

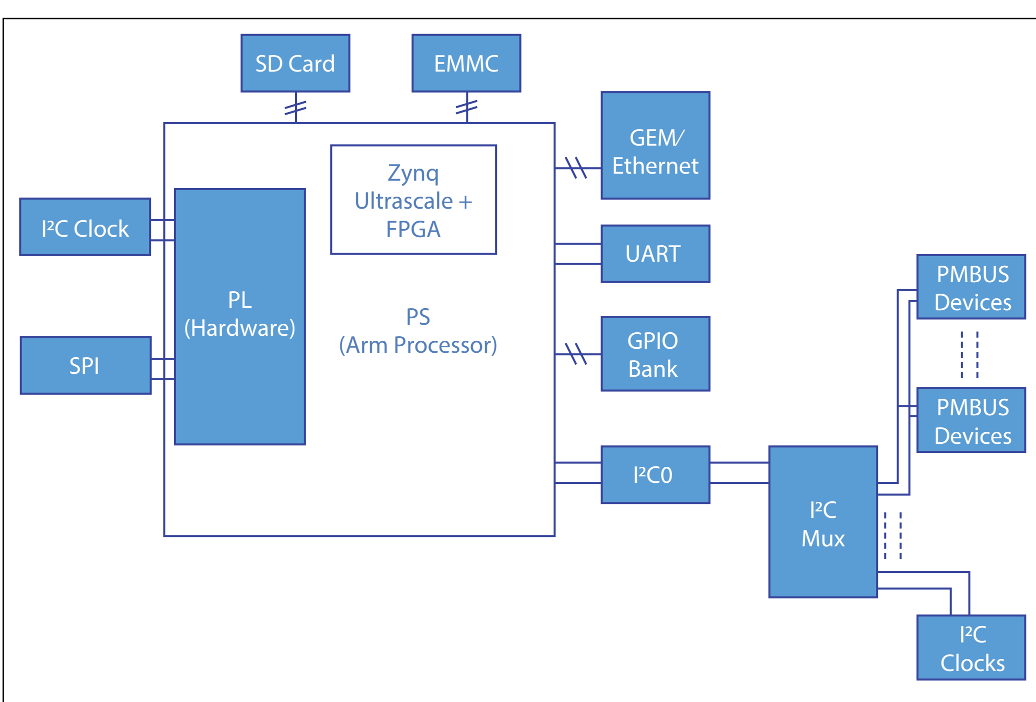


FIGURE 1
Typical block diagram of a system controller

Ethernet is an essential part of the system controller because it helps to control the board from the distance location and enables you to work with the board remotely. UART is a critical element for debugging. The UART is an essential component because it enables you to work with the board locally by connecting the PC to the board via the FTDI circuitry of the UART using a USB Type-A or Type-B cable.

The system controller we're building here is designed to run Linux onboard. In other words, the system controller will be able to boot up Linux and have Linux perform all of the controller's operations. In order to boot Linux—or to transfer any information to the Linux OS—we can use an SD card or an onboard EMMC drive that can boot the processor as well as store information.

Complex platforms bring with them the need to have multiple knobs to control. Power management and clock control are the major knobs to be controlled. Generally, these knobs are all I²C devices and can be connected to a single bus by adjusting the pull up resistors. Using the I²C mux, the number of devices per bus can also be increased. A number of GPIO banks are also necessary. These GPIOs help to provide enable/disable signals, control signals, control LED representations and so forth.

BRING OUT THE TOOLS

Now that the block diagram is defined, we now need to need to understand the overall system requirements and plan the design. To make this design possible, we made use of the PetaLinux tool from Xilinx to create a bootable image to be loaded which has Xilinx board support package. In Part 2 of this article, we'll discuss PetaLinux and how to use it to create the bootable image. Once booted, the Linux system then probes all the devices and enables them. Apart from that, it will also perform power management and clock management using PMBUS protocol and I²C protocol.

Other miscellaneous items such as EEPROM, SPI LCD and GPIOs can all be controlled using standalone applications dumped in the system controller Linux image. All these software aspects will be discussed in detail in Part 2.

Once the overall mapping of the peripherals is done, it's time to create the hardware design of the system controller using Xilinx's Vivado tool. Vivado is a hardware design tool that lets you not only design own IP (intellectual property) blocks, but use existing IPs and connect the blocks using the interactive GUI. The tool can be used to create a "bit" file and an "hdf"—the hardware design file. These two files are necessary to create the Linux image using the file's Xilinx Board Support Package information.

HARDWARE DESIGN

The Vivado tool gives users a visual representation of what the hardware looks like and how it's going to map to the actual device. If you're not already familiar with Vivado, we recommend you take a look at the Vivado the tool guide [2] before reading through this section.

When you first open the Vivado tool, a good first step is to drag and drop the Zynq

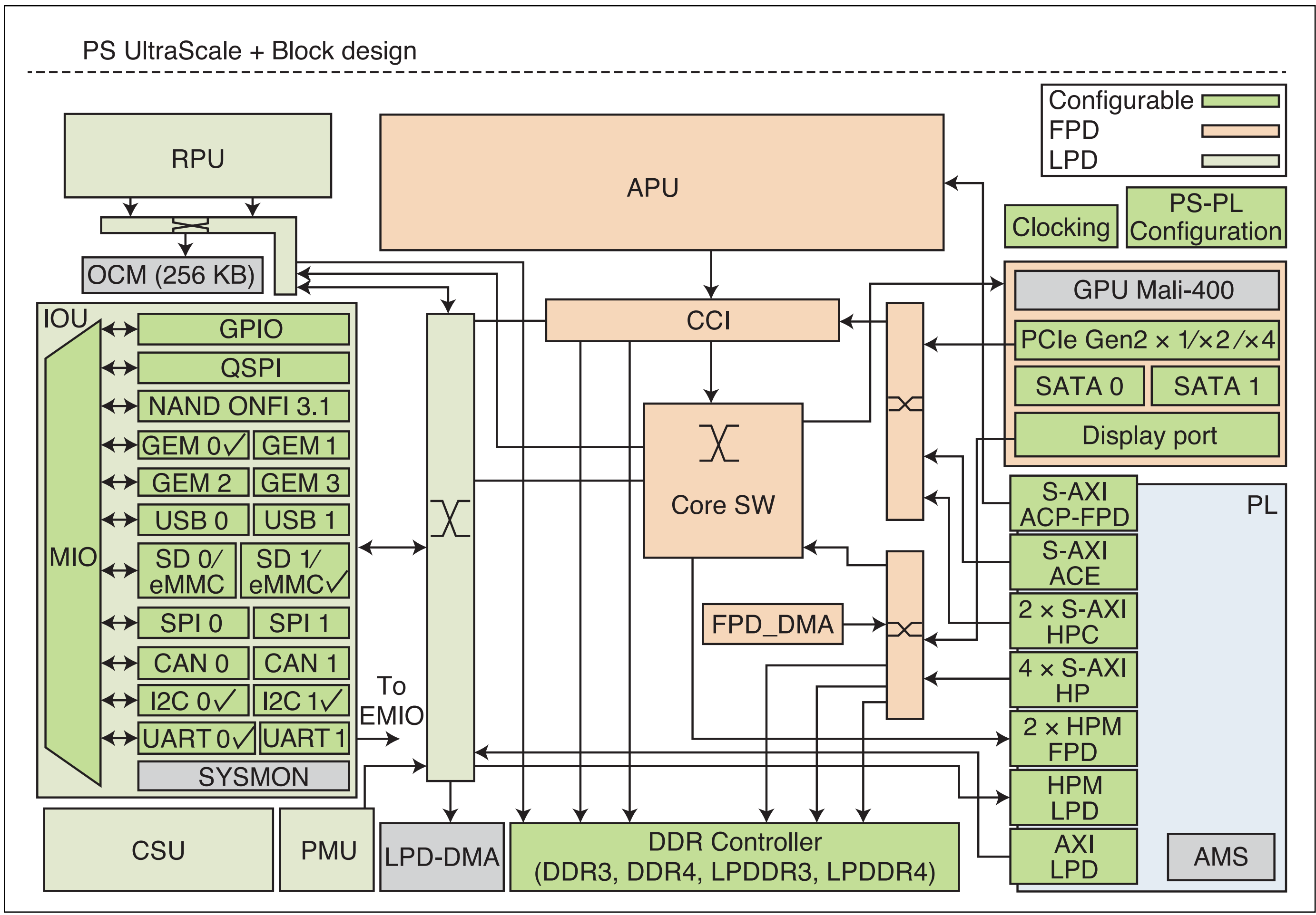


FIGURE 2
Block diagram of the Zynq MP as seen from Vivado tool

Ultrascale+ processor into the drawing window. That processor is the heart of the project, which will connect to all the other peripherals. When you double click that processor block of the FPGA, you can see the overall block diagram of the FPGA. At this point, the entire device is reconfigurable. Given that power is a major concern, it's always a good idea to enable only those blocks that are required for the design and disable the rest of the blocks. Doing that not only consumes less current but also reduces the compilation time. The more hardware you enable, the more time the software will take to create a netlist.

Figure 2 shows the block diagram of the Zynq MP block. Here, we enable Ethernet, SD, eMMC, UART I^{2}C and GPIOs. You could also add Soft IP, which will then get routed through the AXI interface to the IP. **Figure 3** shows the hardware design for the system controller that we talked about in the previous section. Here we have an AXI I^{2}C block that comes from the PL side. We have an AXI interconnect in between that handles all the addressing and clocks—along with signaling to prevent data loss. Because most of the blocks are on the PS side (Figure 2), they won't be visible in the front-end GUI of Vivado.

Once this is done, we need to write the constraints for the design. These include specifying the clock max, assigning pins to the interfaces, setting the default state of the pins and so on. All that information goes into the .xdc file. Here is an example of

ABOUT THE AUTHORS

Nishant Mittal and Manoj Khandelwal are both Systems Engineers with Xilinx in Bangalore, India.

For detailed article references and additional resources go to:
www.circuitcellar.com/article-materials
References [1] and [2[as marked in the article can be found there.

RESOURCES

Xilinx | www.xilinx.com

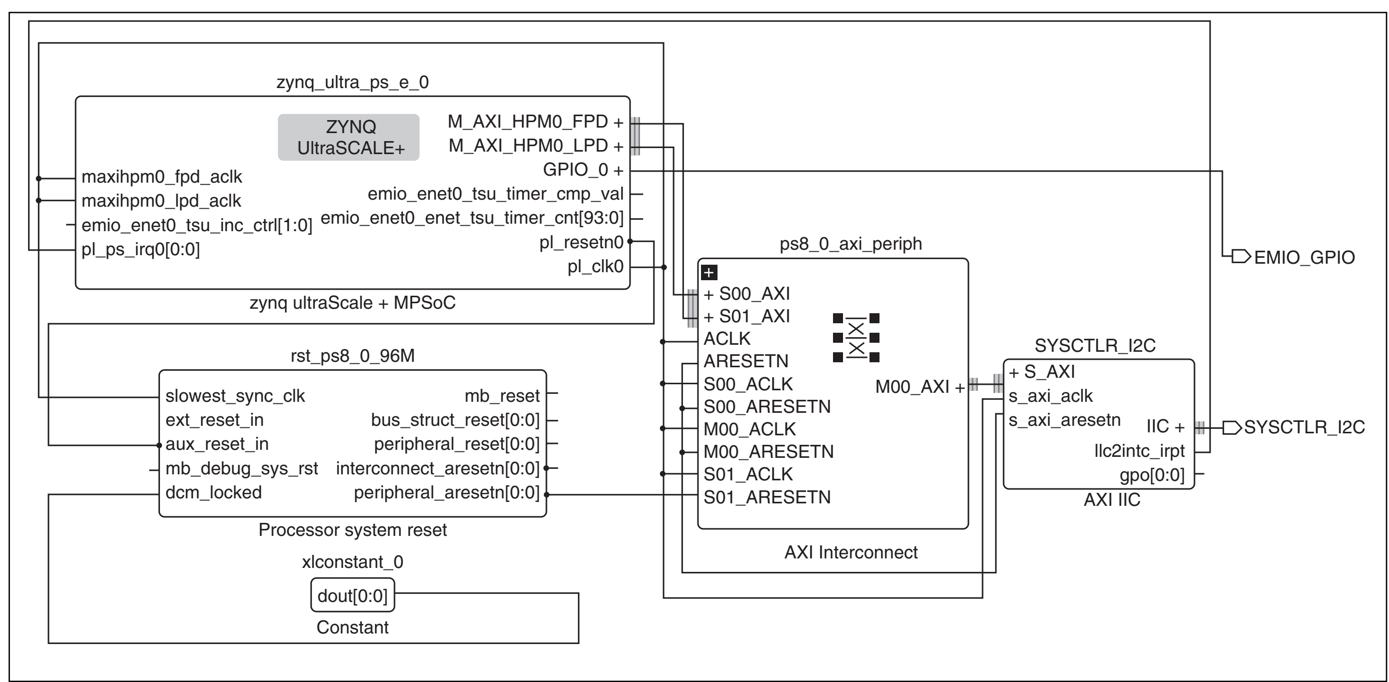


FIGURE 3
Front end of the hardware design in Vivado

some of the constraint properties:

```
set_property PACKAGE_PIN AC14  [get_ports EMIO_GPIO_tri_io[0] ]
set_property PACKAGE_PIN AC13  [get_ports EMIO_GPIO_tri_io[1] ]
set_property PACKAGE_PIN AA13  [get_ports EMIO_GPIO_tri_io[2] ]
set_property PACKAGE_PIN AB13  [get_ports EMIO_GPIO_tri_io[3] ]
set_property PACKAGE_PIN AB15  [get_ports EMIO_GPIO_tri_io[4] ]
set_property PACKAGE_PIN AG13  [get_ports SYSCTLR_SI570_scl_io]
set_property PACKAGE_PIN AH13  [get_ports SYSCTLR_SI570_sda_io]
```

We see that `AG13` and `AH13` are declared as `scl` and `sda` for `SI570` which is the clock frequency generator. `AA13`, `AC13`, `AC14`, `AB13` and `AB15` are declared as tristate GPIOs. Similarly, you can declare your own set of GPIOs based on the platform connections to the system controller. At this stage, the design is ready. Now it can be compiled to generate the .bit and .hdf files. If the design fails, you can use gate level synthesis to understand the reason for failure. That can root out anything from a timing violation to some messy connection. Now that the design is ready, the next step is to bring the design to life using various software tools. Note that system controller need not be present on the same platform board. It can be a separate board if desired, depending on the budget or other user requirements.

CONCLUSION

In this article, we explored the features of system controller and gained an understanding of how it can be useful from various design perspectives. We also learned how to design the hardware part of the system controller using the Vivado toolchain. In Part 2 next month, we'll take a deep dive into the software side of the design, and look at how to bring alive the hardware we designed in this part.

SPECIAL FEATURE

System Solutions Accelerate Drone Development

Fast Track to Flight

By ***Jeff Child,***
Editor-in-Chief

Consumer and commercial drones pose a number of tricky design challenges. Technology vendors have made things somewhat easier over the past year, offering a variety of system-oriented platforms and tools—even including complete development kits.

The development of consumer and commercial drones continues to be a dynamic segment of the embedded systems industry. Faced with severe limits on size, weight and power, drone designers need to be careful with how they choose each and every electronic component. Meanwhile, huge opportunities abound for drone platforms that can pack in high levels of compute processing along with advanced cameras and sensor suites.

Fortunately, drone developers don't have to start from scratch. A rich set of resources are available including board-level solutions, payload subsystems and development kits and even complete reference designs. Over the last 12 months, new solutions along those lines continue to roll out from a variety of vendors ranging from processor companies to drone vendors themselves.

SYSTEM MODULE SOLUTION

Exemplifying those trends, in October Intrinsyc announced its tiny Open-Q uSOM module. The new Open-Q 845 uSOM is a 50mm × 25mm mini-module is based on Qualcomm's Snapdragon 845 SoC (**Figure 1**). It's supported by a Mini-ITX form-factor Open-Q 845 μSOM development kit. The module is designed for advanced robotics, drones and embedded IoT devices requiring the latest on-device AI powers, says Intrinsyc. It runs the Android 9 Pie OS, with a promise to upgrade to the latest Android 10 by 2Q 2020. The module is also supported by a Yocto-based Linux image that is similarly based on Linux kernel 4.9.

Aside from the Open-Q 845 HDK for mobile phones released in 2018, the 8-core, 10nm-fabricated Snapdragon 845 SoC has appeared on the Robotics RB3 Platform from Qualcomm and Thundercomm, which is built around a DragonBoard 845c SBC that has yet to be released separately. More on the RB3 later in this article.

The Open-Q 845 uSOM module ships with 4GB or 6GB dual-channel LPDDR4x SDRAM at 1866MHz, as well as 32GB or 64GB UFS flash. There's also a 2.4/5GHz 802.11a/b/g/n/ac wotj 2×2 MU-MIMO (Qualcomm WCN3990) with a 5GHz external PA and U.FL antenna connector. A Bluetooth 5.x radio is also included. Media interfaces include DisplayPort v1.4 with USB Type-C support for up to 4K60 and 2x 4-lane MIPI-DSI D-PHY 1.2 at up to 3840x2400 10-bit

60fps. Camera interfaces include 3x 4-lane MIPI-CSI and a separate 2-lane MIPI-CSI link.

The development kit for the Open-Q 845 µSOM is built around a 170m × 170mm carrier board. There's also an optional smartphone sized touchscreen and 13-Mpixel camera. The Open-Q 845 µSOM Development Kit carrier runs on 12V/3A power via an included adapter and can also operate on a user-supplied Li-Ion battery.

The board provides a USB 3.1 Type-C port with DP and USB support and there are connectors for all the MIPI-DSI and -CSI interfaces mentioned above. Audio features include the WCD9340 codec, a 3.5mm audio combo jack and analog and digital audio I/O headers. The carrier has a microSD slot, a USB 3.1 host port and a PCIe Gen3 interface. There are also headers for UART, I²C, SPI and configurable GPIOs. Dual PCB antennas are also available.

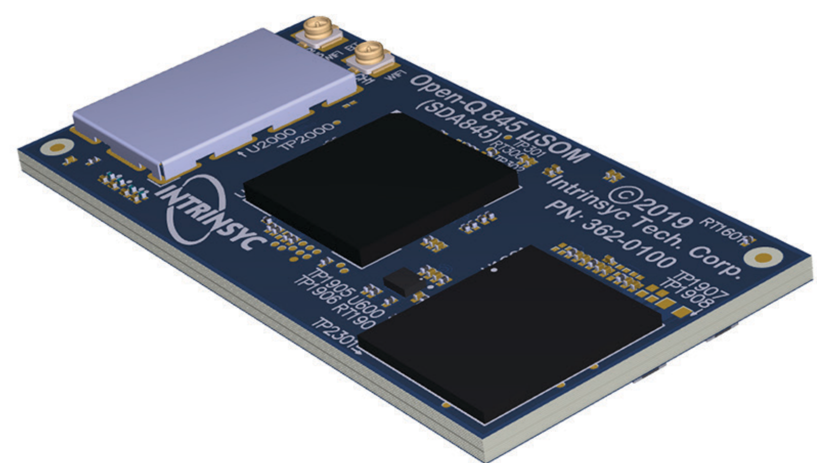


FIGURE 1
Based on Qualcomm's Snapdragon 845 SoC, the tiny Open-Q uSOM is a 50mm x 25mm mini-module designed for advanced robotics drones and embedded IoT devices requiring the latest on-device AI powers.

SMALLEST BREADCRUMB

Among the most compelling advances in commercial drone usage has been the integration of mesh-networks to enable drone communications. Rajant offers a technology solution along those lines. Using a combination of wireless network nodes that Rajant calls Breadcrumbs and its InstaMesh networking software, Rajant's Kinetic Mesh networks employ any-node to any-node capabilities to continuously and instantaneously route data via the best available traffic path and frequency—for any number of nodes, all with extremely low overhead. Rajant BreadCrumbs can communicate with any Wi-Fi or Ethernet-connected device to deliver low-latency, high-throughput data, voice and video applications across the meshed, self-healing network.

In November, Rajant released its latest BreadCrumb product, the DX2. The DX2 is Rajant's smallest and lightest BreadCrumb, forming a mesh network when used in conjunction with its LX5, ME4 and ES1 models, which operate using Rajant's proprietary InstaMesh protocol (**Figure 2**). With one transceiver and two external antennas, DX2 is lightweight and has low power consumption depending on transceiver configuration. Encased in a magnesium enclosure, the DX2 weighs 123g making it well suited for lightweight autonomous vehicles, drones and small robots. This very low payload, combined with a pocket-size footprint, makes it a good solution for varying degrees of autonomy and mobility operations as well as high bandwidth communication and data transmission, according to Rajant.

The DX2 has integrated Wi-Fi access point service for compatibility with millions of commercial off-the-shelf (COTS) client devices, such as laptops, tablets, smartphones, IP cameras, sensors and other IP devices. Additionally, a hidden USB connector, to be used for GPS or Tactical Radio over IP (TRoIP), lies behind a rear black rubber plug.

In compatibility with all other Rajant nodes, the DX2 forms a wireless Kinetic Mesh network that maintains continuous connectivity unlike traditional break-before-make infrastructures, says Rajant. Like all other Rajant BreadCrumbs, the DX2 delivers low-latency, high-throughput, fail-proof connectivity for data, voice and video applications, including drone swarms. The DX2 is available in two models, the DX2-24 with 2.4 GHz and DX2-50 with 5.0 GHz.

HIGH-BANDWIDTH COMMS

Focusing on the high-bandwidth side of drone data transfer, Silvus Technologies provides communications solutions for high bandwidth video, C2, health and telemetry

FIGURE 2
The DX2 is Rajant's smallest and lightest BreadCrumb, forming a mesh network when used in conjunction with its LX5, ME4 and ES1 models, which operate using Rajant's proprietary InstaMesh protocol.

FIGURE 3
The Silent Falcon—a solar electric, fixed-wing drone—integrates Silvus Technologies' MIMO MANET Streamcaster communications systems in its drone systems including its new SF ATAK Field Observer Kit.

data. In September, Silvus announced a partnership with Silent Falcon UAS Technologies, manufacturer of the Silent Falcon, a solar electric, fixed wing, long endurance, long range drone (**Figure 3**). The Silent Falcon drone integrates Silvus' advanced technology MIMO MANET Streamcaster communications systems in its drone systems including its new SF ATAK Field Observer Kit.

Introduced in 2019, the most recent models of the Streamcaster radios are Silvus' Enhanced 4000 series. The new radios provide a user-customizable multilocation switch for loading presets and zeroizing crypto. They have improved connectors and tie-down points for weather caps and feature IP68 enclosures (submersible to 20m). Smart battery technology provides % monitoring. The units also feature FIPS140-2 Level 2 encryption and MANET Interference Avoidance (MAN-IA).

The SC4200E model in the Enhanced series is a 2×2 MIMO radio. It is well suited for use in portable and drone applications where size, weight, power or cost are key. The unit provides up to 4W of output power (up to 8W effective performance thanks to TX Beamforming). The SC4200E is available in three form factors to suit a variety of applications: Rugged "brick" (externally powered), rugged handheld (with twist-lock battery connector) and non-rugged OEM (for embedding in custom products and sub-systems).

Silent Falcon has previously used the Silvus MIMO MANET communications systems for a wide variety of drone long range commercial applications in oil and gas, pipeline, electric power transmission, mapping and surveying markets. It has also been successfully deployed in intelligence, surveillance and reconnaissance; search and rescue and long-range border patrol missions. It's also been used in extreme environmental conditions while assisting the US Department of Interior in wildfire fighting operations.

Silent Falcon recently introduced its three radio SF TriAntenna Ground Control Station, powered by Silvus Streamcaster components. The system increases the reliability, connectivity and bandwidth of the Silent Falcon system. The comm system's capabilities have been further enhanced by the addition of the SF ATAK Field Observer Kit, a small, portable kit that provides live streaming videos with map overlays on tablets and smartphones to operators on the ground who need this vital information in real time.

SNAPS AND MANIFOLD 2

Drones like DJI's Phantom and Matrice models embed flight controllers that run a proprietary operating system. But, in 2015, the company announced a Manifold development computer for its Matrice 100 drone that runs Ubuntu on a Nvidia Tegra K1. In June 2019, DJI unveiled a more powerful Manifold 2 computer with a choice of Nvidia Jetson TX2 and Intel Core i7-8550U processors (**Figure 4**). Canonical followed up by announcing that, not only will Ubuntu 16.04 return as the pre-installed OS for the device, but that it will include support for Ubuntu snaps application packages.

Ubuntu snaps are containerized software packages that work interchangeably across embedded, desktop and cloud-based Ubuntu distributions. Found on embedded Linux devices ranging from LimeSDR boards to Orange Pi PCs, they offer built-in security, automated updates and transaction rollback support. They also come with an online

For detailed article references and additional resources go to:
www.circuitcellar.com/article-materials

RESOURCES
Aerotenna | www.aerotenna.com

Intrinsyc Technologies | www.intrinsyc.com

Nvidia | www.nvidia.com

NXP Semiconductors | www.nxp.com

Qualcomm | www.qualcomm.com

Rajant | www.rajant.com

Silvus Technologies | www.silvustechnologies.com

marketplace for sharing and selling different snaps applications. The Manifold 2 will be the first drone system to offer snaps, which will enable its functionality to be "altered, updated and expanded over time," according to Canonical. Snaps will make it easier to manage large fleets of drones, as well as develop vertical applications that can be shared and modified for other use cases.

Ubuntu offers DJI drone users support for Linux, Nvidia CUDA, OpenCV and ROS (Robot Operating System). The Ubuntu-driven Manifold 2 is well suited for the research and development of professional applications and can access flight data and perform intelligent control and data analysis. The Manifold 2 can be integrated on DJI enterprise drones including the Matrice 210 series and Matrice 600 series, as well as its separately available N3 Flight Controller and A3 Flight Controller. The computer can process complex image data onboard the drone and get results immediately and can program drones to fly autonomously while identifying objects and avoiding obstacles, says DJI.

The Manifold 2 can act either as a companion computer or as a control computer over the flight controller. The system can be integrated into the drone's internal systems and sensors using DJI's software development kit. The Manifold 2 offers users a choice of two processing platforms, both of which run Ubuntu 16.04 with snaps. The first is the "GPU Model" (Manifold2-G) with Nvidia's Jetson TX2, which offers a more powerful, hexa-core update to the Manifold 1's Nvidia Tegra K1.

DJI lists different applications for the two models. The GPU Model is said to be designed for AI, object recognition, motion analysis and image processing. The CPU Model is for autonomous flight, real-time data analysis, ground station connectivity and robotics. Both Manifold 2 versions have a -25°C to 45°C tolerant, 91mm × 61mm × 35mm enclosure, down from 110mm × 110mm × 26mm on the Manifold 1. Despite the smaller footprint, the new models are heavier than the under 200g original. The Jetson TX2-based GPU model weighs in at 230g while the Coffee Lake-based CPU Model is 205g.

FIGURE 4
The Manifold 2 will be the first drone system to offer snaps, which will enable its functionality to be altered, updated and expanded over time.

COMPLETE DRONE DEV KIT

We've discussed several drone development kits in *Circuit Cellar* in recent years. These kinds of kits provide all the components needed to get a drone platform up and running. An example is the Smart Drone Development Platform from Aerotenna. The kit is equipped with microwave radar collision-avoidance sensors, a radar altimeter

FIGURE 5
The Smart Drone Development Platform from Aerotenna is equipped with microwave radar collision-avoidance sensors, a radar altimeter and an FPGA-based flight controller.

SPECIAL FEATURE

and an FPGA-based flight controller (**Figure 5**).

The kit's OcPoC Zynq Mini Flight Controller is an FPGA-based flight controller capable of triple redundant GPS, compass and IMU. The unit is pre-loaded with the PX4 and Ardupilot flight control stacks. PX4 is the largest commercially deployed open source flight stack and supports contemporary airframe architectures including VTOL aircraft, multicopter and rover profile. The µLanding radar altimeter can perform in all weather conditions and challenging terrains and has maximum altitude range of 150m with and altitude accuracy of 2cm. Its update rate is 766Hz (every 1.31ms).

The development kit also includes three µSharp-Patch collision avoidance radar sensors. These sensors scan the front, left and right side of the vehicle, detecting and locating obstacles on the horizon quickly and reliably and a maximum range of 120m. A pre-assembled quadcopter, carbon fiber airframe is provided in the kit. It includes GPS/compass, foldable arms for ease of transport and modular component design for simple maintenance and repair. It can do flight times up to 50 minutes (16000mAh battery, no payload). Total weight with airframe, pre-assembled flight controller and sensors is 1.9kg. Maximum takeoff weight with battery is 4.0kg.

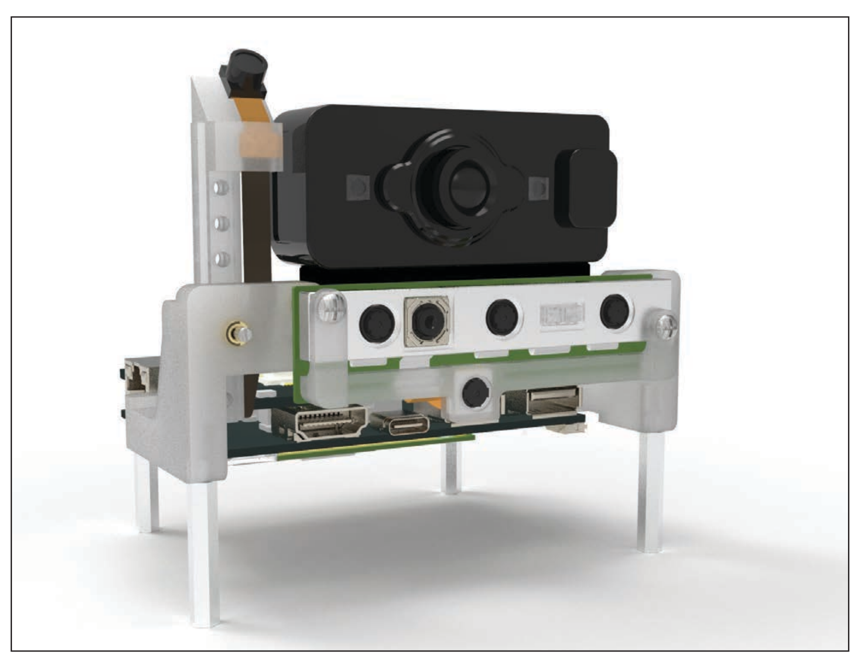


FIGURE 6
The RB3 platform includes an octa-core Snapdragon 845 via a "DragonBoard 845c" 96Boards SBC and tracking cameras. It integrates key capabilities such as high-performance heterogeneous computing, 4G/LTE connectivity including CBRS support for private LTE networks, advanced security and Wi-Fi connectivity.

DEV KIT RUNS ROS

Among the drone development kits introduced in 2019 was the Robotics RB3 Platform co-developed by Qualcomm and Thundercomm (**Figure 6**). The platform includes an octa-core Snapdragon 845 via a new "DragonBoard 845c" 96Boards SBC and tracking cameras. While the platform appears to be marketed toward terrestrial robots, Qualcomm told us that it's also suited for developing drones.

The RB3 platform integrates key capabilities such as high-performance heterogeneous computing, 4G/LTE connectivity including CBRS support for private LTE networks, a Qualcomm AI Engine for on-device machine learning and computer vision, hi-fidelity sensor processing for perception, odometry for localization, mapping and navigation, advanced security and Wi-Fi connectivity. Support is also planned for 5G connectivity.

The platform currently supports Linux and Robot Operating System (ROS), while also including support for the Qualcomm Neural Processing software development kit (SDK) for advanced on-device AI, the Qualcomm Computer Vision Suite, the Qualcomm Hexagon DSP SDK and Amazon's AWS RoboMaker, with plans for Ubuntu Linux support.

The platform's hardware development kit contains the new purpose-built robotics-focused DragonBoard 845c development board, based on the Qualcomm SDA/SDM845 SoC and compliant with the 96Boards open hardware specification to support a broad range of mezzanine-board expansions. Optional elements for the kit include a connectivity board; an image camera for superb hi-res photo, 4K video capture and AI-assisted detection and recognition of people and objects; a tracking camera for path planning and obstacle avoidance using visual simultaneous localization and mapping (vSLAM); a stereo camera for navigation; and a time-of-flight camera for people, gesture and object detection even in low light conditions.

KIT FOR DRONE CHALLENGE

There's been a history of processor vendors providing drone development kits—Intel and Qualcomm, for example. NXP Semiconductors for its part, has put a twist on this trend by making a drone development kit part of an annual drone development contest. Called the HoverGames Challenges, participants use NXP's HoverGames drone development kit. The hardware and software of the developer kit is open, flexible and modular and includes

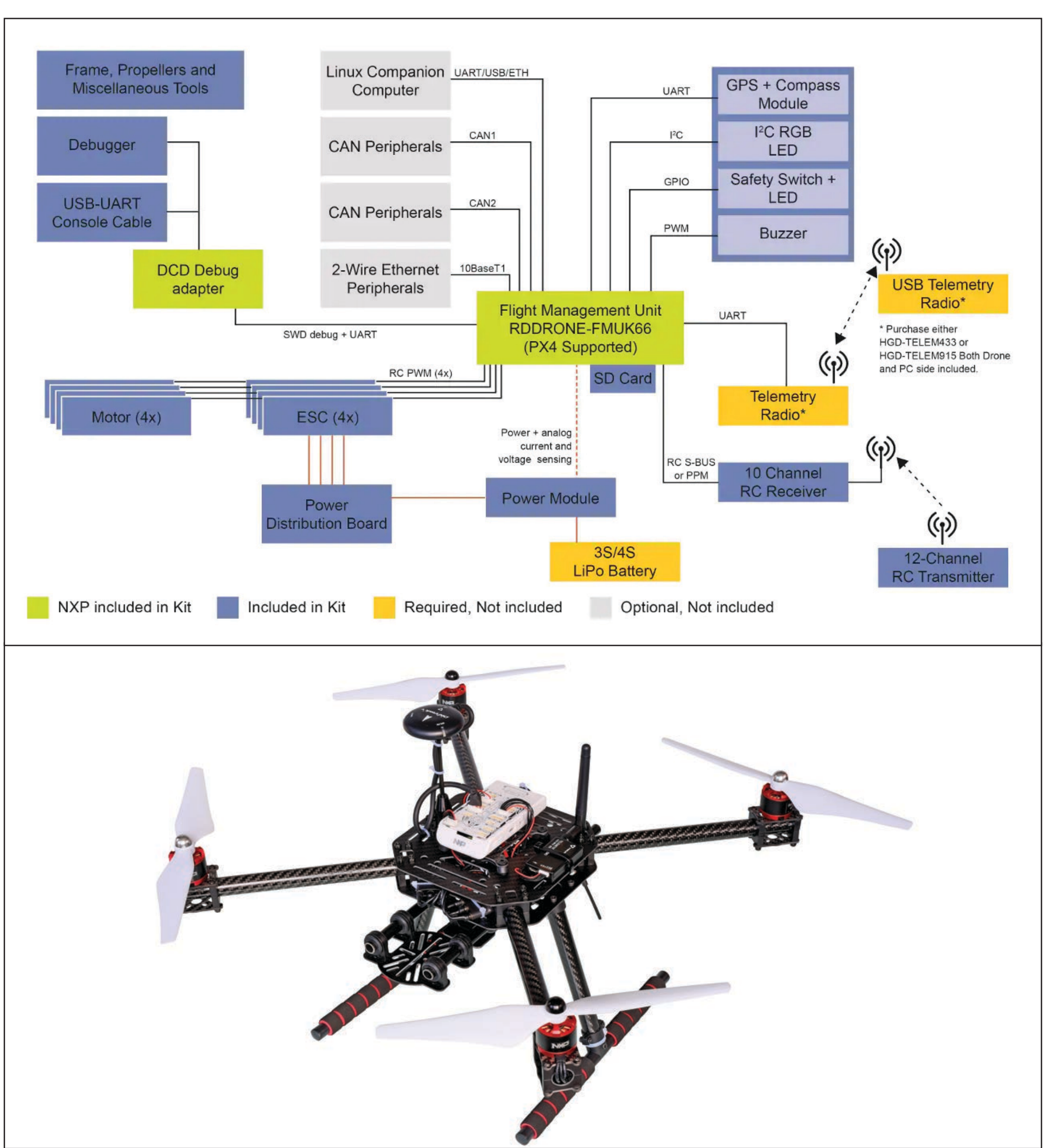


FIGURE 7
Top: The HoverGames KIT-HGDRONEK66 kit provides the mechanical and other components needed to evaluate the RDDRONE-FMUK66 flight management unit and adds BLDC motor control capabilities and a mechanical platform, which it can be mounted on. *Bottom:* an assembled HoverGames RDRONE drone.

professional, automotive and industrial-grade components enabled by the PX4 flight stack.

The HoverGames KIT-HGDRONEK66 kit (**Figure 7**) provides the mechanical and other components needed to evaluate the RDDRONE-FMUK66 flight management unit and adds BLDC motor control capabilities and a mechanical platform, on which it can be mounted. This developer kit may be used as part of and contains the components needed for the HoverGames coding challenges. NXP points out that this is a professional developer kit, not a complete functional system and includes no software. The flight management unit (FMU) is supported by the business-friendly open source PX4 flight stack. In addition, a separate suitable hobby-type LiPo battery and country-specific telemetry radio will be required.

When assembled, the frame has appropriate the additional space necessary to mount other components such as an adapter for Rapid IoT, NXP Freedom boards, or a companion computer such as i.MX 8M Mini to be used as a vision processor running Linux and ROS. The HoverGames drone and rover development platform is very flexible, fully open for development of robotics, control algorithms, security networking and communications protocols and can include another add-on component, companion computer, software or associated solutions.

Today's quadcopter style consumer and commercial drones couldn't exist without today's high levels of chip integration. As developers push for more autonomous operations and AI aboard drones, they'll continue to look toward SoC-based solutions to offer improved functionally without added size and weight. Fortunately, technologies and solutions such as those covered in this article can help drone system developers to get to market—and to flight—faster.

TECH SPOTLIGHT

Analog ICs Boast Battery Management Innovations

Perfecting Power

By *Jeff Child,*
Editor-in-Chief

FIGURE 1
The MAX17301 and the MAX17311 fuel gauge ICs offer configurable settings for battery safety and allow fine tuning of voltage and current thresholds based on various temperature zones.

Boosting battery life and efficiency is a major goal for many embedded systems. Analog IC vendors are smoothing the way with innovative chips for monitoring, controlling and charging batteries.

Managing battery power is a critical function for all sorts of battery-powered systems, including power tools, wearable electronics, IoT edge devices and electric vehicles. Innovations in power management ICs, fuel-gauge ICs, battery monitoring ICs and more are helping to provide improved power efficiency for diverse applications.

There are many facets to managing batteries in embedded systems. To meet the ever-present goal of extending battery lifetimes and battery efficiencies requires solutions for monitoring and charging batteries, as well as efficient power conversion devices. Over the past 12 months, analog ICs vendors have rolled out several innovative solutions both for portable, battery-powered systems and for the particular needs for electric vehicle battery management.

FUEL GAUGE ICs

Along those lines, in August Maxim Integrated announced fuel gauge ICs that company claims offer the most configurable settings for battery safety in the industry and uniquely allow fine tuning of voltage and current thresholds based on various temperature zones. The newest 1-cell, pack-side ICs in this portfolio are the MAX17301 and the MAX17311 (**Figure 1**). These ICs also offer a secondary protection scheme in case the primary protection fails. This secondary protection scheme permanently disables the battery by overriding a secondary protector or blowing a fuse in severe fault conditions.

All ICs in the family are equipped with Maxim's patented ModelGauge m5 EZ algorithm that delivers highest state-of-charge (SOC) accuracy that on average offers 40% better accuracy than competitive offerings and eliminates the need for battery characterization. These fuel gauges also offer the industry's lowest quiescent current (IQ)—up to 80% lower than the nearest competitor according to Maxim, and feature SHA-256 authentication to safeguard the systems from counterfeit batteries.

Conventional battery protectors monitor voltage and current, and in some cases include temperature monitoring, says Maxim. These options make the system vulnerable to unexpected crashes because battery SOC isn't factored in when triggering an undervoltage

cut-off decision. The market lacks a solution that allows deeper configuration of voltage or current thresholds based on multiple temperature environments.

Maxim's devices provide advanced battery protection to ensure safe charging and discharging in a wide range of applications with 2-level Li-ion protector control for abnormal voltage, current and temperature conditions. The ICs protect against counterfeiting and cloning with SHA-256 authentication and provide unique as well as dynamic keys for every battery.

To enable high accuracy, the chips offer cycle+ age forecasting that provides easy-to-understand prediction of remaining battery life for battery replacement planning or to control fast-charging. Battery life logging stores the history of operating conditions experienced by the pack over its lifetime. Support for long product shelf-life and runtime is served by an operating IQ of 24µA active/18µA low power with protector FETs "on" and 7µA with protector FETs "off."

BATTERY CHARGER IC

With its focus on the charging side of battery management, in September Texas Instruments (TI) introduced a switching battery charger IC that supports a termination current of 20mA. Compared to competing devices, which typically support a termination current higher than 60mA, TI's BQ25619 enables 7% higher battery capacity and longer runtime, says TI. The BQ25619 charger also delivers three-in-one boost converter integration and ultra-fast charging, offering 95% efficiency at a 4.6V and 0.5A output (**Figure 2**). Additionally, with the industry's lowest quiescent current, the new charger can double the shelf life of ready-to-use electronics.

The BQ25619 charger is designed to help engineers design more efficiently for small medical and personal electronics applications such as hearing aids, earbuds and wireless charging cases, IP network cameras, patient monitoring devices and personal care applications.

An ultra-low termination current of 20mA increases battery capacity and runtime by up to 7%. The BQ25619's settable top-off timer further increases run time, enabling users to charge their devices less frequently. The BQ25619 reduces battery leakage down to 6µA in ship mode, which conserves battery energy to double the shelf life for the device. While in battery-only operation, the device consumes only 10µA, to support standby systems.

The BQ25619 includes integrated charge, boost converter and voltage protection to support efficient design for space-constrained applications and eliminate the external inductor required by previous-generation charger ICs. Due to its integrated bidirectional buck or boost topology, the BQ25619's charging and discharging capabilities require just a single power device. The device is offered in a 24-pin wafer quad flatpack no-lead (WQFN) package. The 30-pin BQ25618, with similar features, is offered in a smaller wafer chip-scale package (WCSP).

WIRELESS CHARGING IC

Many wearable devices aren't suited to be powered by replaceable batteries. As a result, they typically need to be recharged. Wireless (cordless) battery charging is beginning to take hold as a solution. Feeding such needs, Analog Devices offers its LTC4126 as an expansion of its offerings in wireless battery charging. The LTC4126 combines a wireless powered battery charger for Li-Ion cells with a high efficiency multi-mode charge pump DC-DC converter, providing a regulated 1.2V output at up to 60 mA (**Figure 3**).

Charging with the LTC4126 allows for a completely sealed end product without wires or connectors and eliminates the need to constantly replace non-rechargeable (primary) batteries. The efficient 1.2V charge pump output features pushbutton on/off control and can directly power the end product's ASIC. This greatly simplifies the system solution and reduces the number of necessary external components. The device is ideal for space-constrained low power Li-Ion cell powered wearables such as hearing aids, medical smart patches, wireless headsets and IoT devices.

FIGURE 2
The BQ25619 charger delivers three-in-one boost converter integration and ultra-fast charging, offering 95% efficiency at a 4.6V and 0.5A output.

TECH SPOTLIGHT

The LTC4126, with its input power management circuitry, rectifies AC power from a wireless power receiver coil and generates a 2.7V to 5.5V input rail to power a full-featured constant-current/constant-voltage battery charger. Features of the battery charger include a pin selectable charge voltage of 4.2V or 4.35V, 7.5mA charge current, automatic recharge, battery temperature monitoring via an NTC pin, and an onboard 6-hour safety charge termination timer.

Low-battery protection disconnects the battery from all loads when the battery voltage is below 3.0V. The LTC4126's charge pump switching frequency is set to 50kHz/75kHz to keep switching noise out of the audible range, ideal for audio related applications such as hearing aids and wireless headsets. The IC is housed in a compact, low profile (0.74 mm) 12-lead 2mm × 2mm LQFN package. The device is guaranteed for operation from –20°C to 85°C in E-grade.

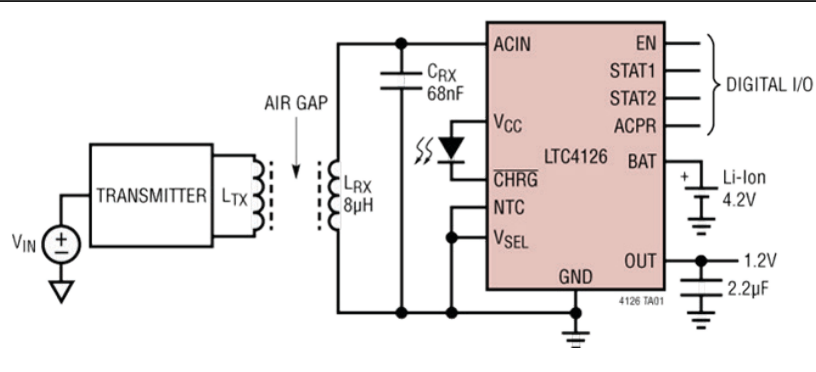


FIGURE 3
The LTC4126 combines a wireless powered battery charger for Li-Ion cells with a high efficiency multi-mode charge pump DC-DC converter, providing a regulated 1.2V output at up to 60mA.

FIGURE 4
The ISL78714 provides accurate cell voltage and temperature monitoring, along with cell balancing and extensive system diagnostics to protect 14-cell Li-ion battery packs while maximizing driving time and range for hybrid and electric vehicles.

SOLUTION FOR 14 Li-Ion CELLS

Electric and hybrid vehicles have very special requirements when it comes to managing their battery subsystems. Feeding those needs, Renesas Electronics in August announced its fourth-generation Li-Ion battery management IC that offers high lifetime accuracy. The ISL78714 provides accurate cell voltage and temperature monitoring, along with cell balancing and extensive system diagnostics to protect 14-cell Li-Ion battery packs while maximizing driving time and range for hybrid and electric vehicles (**Figure 4**).

The ISL78714 monitors and balances up to 14 series-connected cells with ±2mV accuracy across automotive temperature ranges, letting system designers make informed decisions based on absolute voltage levels. The ISL78714 includes a precision 14-bit analog-to-digital converter and associated data acquisition circuitry. The device also offers up to six external temperature inputs (two available from GPIOs) and includes fault detection and diagnostics for all key internal functions.

The ISL78714 meets the stringent reliability and performance requirements of battery pack systems for all electric vehicle variants, with safety features, enabling automotive manufacturers to achieve the ISO 26262 automotive safety integrity level (ASIL D). In addition, the ISL78714 monitors and reads back over/under voltage, temperature, open wire conditions, and fault status for 112 cells in less than 10ms, or 70 cells in 6.5ms.

Multiple ISL78714s can be connected together via a proprietary daisy chain that supports systems up to 420 cells (30 ICs) that provide industry-leading transient and EMC/EMI immunity, exceeding automotive requirements. The ISL78714's daisy-chain architecture uses low-cost capacitive or transformer isolation, or a combination of both, with twisted pair wiring to stack multiple

RESOURCES

Analog Devices | www.analog.com

Maxim Integrated | www.maximintegrated.com

Renesas Electronics | www.renesas.com

Texas Instruments | www.ti.com

battery packs together while protecting against hot plug and high voltage transients. A watchdog timer automatically shuts down a daisy-chained IC if communication is lost with the master MCU. Mass production quantities of the ISL78714 Li-ion battery management IC are available now in a 64-lead TQFP package.

ELECTRIC VEHICLE DESIGN WIN

In December, Analog Devices (ADI) announced that Rimac Automobili is planning to incorporate ADI's precision battery management system (BMS) ICs into Rimac's BMS. ADI's technology provides Rimac's BMS with the ability to extract maximum energy and capacity out of its batteries by calculating reliable SOC and other battery parameters at any given time, according to ADI.

The Rimac C_Two is a fully electric hypercar capable of speeds of up to 258 miles per hour. With 1,914 horsepower under the hood, the C_Two accelerates 0 to 60 mph in 1.85 seconds and 0 to 186 mph in 11.8 seconds. To support these high-performance outputs, the Rimac team designs and engineers superior underlying technologies, such as electric drivetrain and battery packs.

BMS technology acts as the "brains" behind battery packs by managing the output, charging and discharging as well as providing precision measurements during vehicle operation. A BMS also provides vital safeguards to protect the battery from damage. A battery pack consists of groups of individual battery cells that work seamlessly together to deliver maximum power output to the car. If the cells go out of balance, the cells can get stressed leading to premature charge termination and a reduction in the battery's overall lifetime. ADI's battery management ICs deliver the highly accurate measurements, resulting in safer vehicle operation and maximizing vehicle range per charge.

ASIL-D COMPLIANT IC

Safety standards compliance is a key concern in electric vehicles. Automotive designers can now achieve ASIL-D compliance for automotive applications using just a single chip for a safer, more cost-effective battery management system with the MAX17853 battery monitor IC from Maxim Integrated (**Figure 5**). Targeting mid-to-large cell count configurations for automotive applications, such as battery packs for electric and hybrid vehicles, MAX17853's flexible architecture called Flexpack enables engineers to rapidly make changes to their module configurations to quickly respond to market demands.

Achieving safety compliance in automotive applications can require adding redundant components to the system. Maxim claims that the MAX17853 is the only ASIL-D-compliant IC for mid-to-large cell count configurations, enabling engineers to create a system that meets the highest level of safety for voltage, temperature and communication. Also contributing to higher safety is the device's advanced battery cell balancing system, which automatically balances each cell by time and voltage to minimize risk of overcharging.

System developers can achieve all this without adding extra components such as redundant comparators to help achieve smaller form factors, says Maxim. In addition, the MAX17853 reduces system bills of materials (BOM) cost by up to 35% compared to competitive solutions to allow the customer to achieve lower overall cost for their BMS solution.

Flexibility is also important because engineers typically must design and qualify separate boards and BOMs for each different module configuration. The MAX17853 is the industry's only IC supporting multiple channel configurations (8 to 14 cells) with one board. This enables engineers to reduce design time by up to 50% through reduced validation and qualification time. For example, they can cut their development time and qualification efforts in half by using the same board for 8s and 14s modules.

FIGURE 5
Maxim claims that the MAX17853 is the only ASIL-D-compliant IC for mid-to-large cell count configurations, enabling engineers to create a system that meets the highest level of safety for voltage, temperature and communication.

Datasheet: COM Express Boards

Compact Performance

By ***Jeff Child,***
Editor-in-Chief

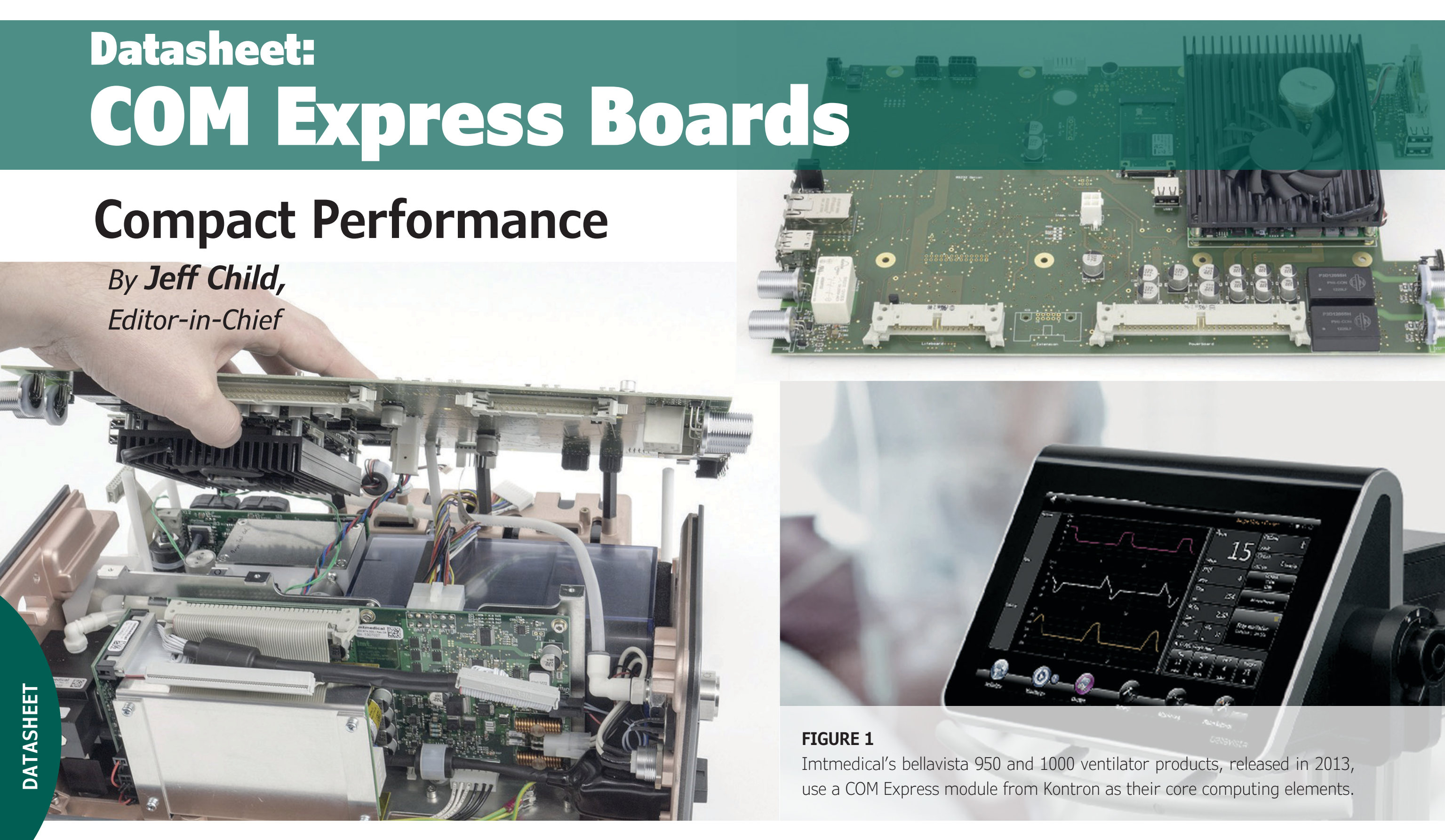


FIGURE 1
Imtmedical's bellavista 950 and 1000 ventilator products, released in 2013, use a COM Express module from Kontron as their core computing elements.

COM Express has emerged as one of the most popular standards-based form factors for embedded systems. COM Express modules serve as a complete computing core that can be upgraded when needed, while the application-specific I/O on the baseboard can remain the same.

COM Express has all the aspects that make a successful embedded board-level form factor: a large ecosystem of vendors that make COM Express boards, an active and innovative standards organization in the form of PICMG (PCI Industrial Computer Manufacturers Group) and a wide application base of engineers hungry to embedded the technology into their systems. As ever more powerful processors emerge, embedded computing modules like COM Express boards will only get more powerful. The approach of a two-board solution—a COM Express module and an I/O baseboard—has caught on while slot-card system architectures have begun to lose favor. According to multiple research reports, the computer-on-module (COM) market is expanding rapidly and is expected to reach over $1 billion by 2022.

In November, the PICMG COM-HPC technical subcommittee approved the pinout of its high-performance Computer-on-Module specification. The new COM-HPC standard is now entering the home stretch for the ratification of version 1.0 of the specification, which is scheduled for the first half of 2020. COM manufacturers and carrier board designers who are active in the COM-HPC workgroup can now embark on their first edge computing designs based on this pre-approved data, with the expectation to bring them to market in time with the launch of new high-end embedded processor generations from Intel and AMD in 2020. PICMG says the new COM-HPC is in parallel to existing COM Express efforts. This effort is intended to complement rather than be a replacement for COM Express.

COM Express is widely used in industrial automation, defense/aerospace, gaming, medical, transportation, IoT and other applications. Here's an example of a medical application using COM Express: In order to optimize emergency patient care, the Swiss IT company Imtmedical, a manufacturer of solutions in medical ventilation technology, used COM Express technology from Kontron for a ventilation system they built in partnership with IMT AG. The resulting bellavista 950 and 1000 products, released in 2013, use a COM Express module from Kontron as their core computing elements (**Figure 1**).

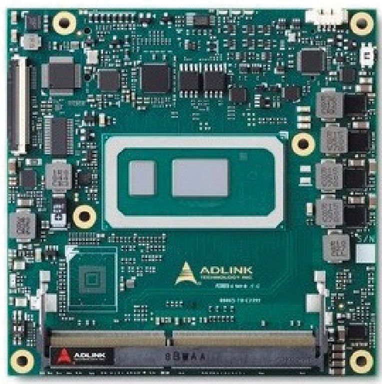
The representative set of COM Express board in this section are limited to products announced in the last 12 months. In the digital version of this article, you can access links to the actual datasheets of each product.

Type 6 COMe Board Targets Edge Computing

Aaeon's COM-CFHB6 is built to the COM Express Type 6 form factor. It features a wide range of processors from the Intel Celeron to Intel Xeon, and the 8th and 9th Generation Intel Core processors (Coffee Lake H/Coffee Lake Refresh). The COM-CFHB6 supports low power 25W processors, well suited for mobile applications, up to 45W 6-core Xeon server CPUs.

- Intel Coffee Lake-H 8th / 9th Ge i3/ i5/i7/ Xeon-E processors
- 3x SODIMM DDR4 2666 memory up to 48GB, ECC support (by SKU, with CM246 PCH)
- Intel I219 Gbit Ethernet
- VGA, 18/24-bit 2ch LVDS or 4-lane eDP, DDI up to 3
- High definition audio interface
- SATA x4, USB2.0 x8, USB 3.0 x4
- PCI-Express [x1] x8, PCI-Express [x16] x1
- GPIO x 8, SMbus, I2C, LPC
- COM Express Basic size, pin-out Type 6, 125mm x 95mm

AAEON
www.aaeon.com



COMe Board Serves up 15W Quad-Core Processors

ADLINK Technology's cExpress-WL modules feature the 8th generation Intel Core and Celeron processors (formerly codenamed Whiskey Lake-U) with up to 4 cores and up to 64GB memory capacity. The cExpress-WL is suited for applications such as data acquisition and analysis, image processing, and 4K video transcoding and streaming at the edge.

- 8th gen quad/dual-core Intel Core processors
- Up to 64GB dual channel non-ECC DDR4 at 2133/2400MHz
- 2x DDI channels, 1x LVDS, one opt. VGA, supports up to 3 independent displays
- Up to eight PCIe lanes, GbE
- Up to 3x SATA 6 Gb/s, four USB 3.1 Gen2 and four USB 2.0
- Supports Smart Embedded Management Agent (SEMA) functions
- Operating temperature: -40°C to +85°C (optional)

ADLINK Technology
www.adlinktech.com

COMe Type 6 Card Sports Intel H-Series Processor

Advantech's high-end SOM-5899 series COM Express Type 6 Module is designed with 8th and 9th Gen Intel Core H-series processors. Compared with previous generations, the SOM-5899 is enhanced with six cores for better multithreaded compute-intensive application performance.

- COM Express R3.0 Type 6 Basic module
- 8th and 9th Gen Intel Core Xeon and i7/i5/i3/Celeron processors
- 2 to 6 Core CPU with up to 96GB Non-ECC memory and up to 48GB ECC memory
- 3x DDI 4k resolution
- 4x USB3.1 Gen 2 (10Gbps) / PEG x16 and 8x PCIe Gen3 (8Gbps)
- 4x SATA III ports and supports AHCI and RAID mode
- Supports Advantech iManager, WISE-PaaS/DeviceOn

Advantech
www.advantech.com

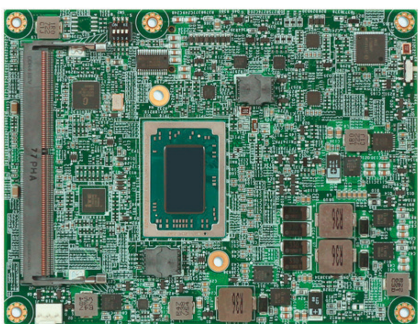
DATASHEET URLS:

AAEON www.aaeon.com/en/p/com-express-modules-com-cfhb6

ADLINK Technology www.adlinktech.com/Products/Computer_on_Modules/COMExpressType6Compact/cExpress-WL

Advantech https://advdownload.advantech.com/productfile/PIS/SOM-5899/file/SOM-5899_5899R_DS(120219)20191202190722.pdf

COM Express Boards

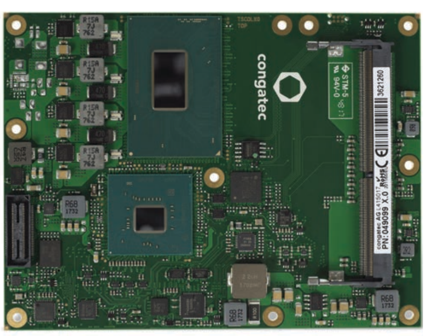


COMe Type 6 Board Features AMD Ryzen Embedded SoC

The PCOM-B701 from American Portwell is a Type 7 COM Express basic (125mm x 95mm) module is designed with the Intel Atom processor C3000 product family (codenamed Denverton). Specifically, the COM Express 3.0 specification's Type 7 pinout, when compared to the Type 6 pinout, trades all the graphics interfaces for up to four 10 GbE ports and a total of 32 PCIe lanes.

- COM Express Type 6 Basic form factor
- Up to 4 Ryzen cores, AMD Radeon GCN compute units
- Supports dual channel ECC DDR4 SO-DIMM horizontal socket (up to 16GB)
- Supports four 4K displays
- Supports USB 2.0/3.0, 1xSATA, 1x PCIe x8 and 4x PCIe x1
- Next-generation AMD secure processor

American Portwell
www.portwell.com



COM Express Card Offers 14 Different Processor Options

The conga-TS370 Type 6 COM Express board from Congatec supports 14 processor variants. In July, the company added ten new variants to the original four. The new ones include four Intel Xeon, three Intel Core, two Intel Celeron and one Intel Pentium processor—all based on the same Intel microarchitecture (codenamed Coffee Lake H). This enables Congatec to provide all 10 new processors on one COM Express module design.

- 8th gen Intel Core processor with up to 6 Cores
- Intel Xeon processors for data center applications
- Support for USB 3.1 Gen2 with 10Gbit/s
- Intel Optane memory support
- ECC memory support
- Up to 64GB dual channel DDR4 memory

Congatec
www.congatec.com

Type 6 COMe Module Has Extended Temp Support

Eurotech's CPU-162-23 brings the computational performance and RAM capacity of a server to the field. It supports extended temperature range (-40 to +85°C). The soldered-down CPUs and ECC memory further increase reliability in demanding applications. The board provides full integration with Eurotech IoT Edge Framework Everyware Software Framework (ESF), providing native connectivity with many IoT Cloud services

- HPEC and micro server ready
- Intel Pentium and Xeon D-1500 CPUs
- Up to four SO-DIMM sockets for a total of 64GB DDR4 with or without ECC
- 2x 10Gbit Ethernet
- Up to x32 PCIe lanes
- 2x SATA 3.0 ports, 4x USB 3.0 and 4x USB 2.0 ports
- Rugged and fanless

Eurotech
www.eurotech.com

DATASHEET URLS:

American Portwell www.portwell.com/pdf/embedded/MEDM-B603.pdf

Congatec www.congatec.com/fileadmin/user_upload/Documents/Datasheets/conga-TS370.pdf

Eurotech www.eurotech.com/en/products/boards-modules/comexpress/cpu-162-23

AMD Ryzen-Based COMe Cards Support Industrial Temps

The COMe-cVR6 (E2) from Kontron marries the COM Express compact form factor and AMD's Ryzen Embedded V-Series processors. Through the use of consistent COM Express connectors and feature implementation, the COMe-cVR6 is easily exchangeable and offers the most flexibility for engineers designing it into their embedded devices based on individual carrier boards.

- AMD Ryzen Embedded V1000 APUs
- Up to 4 independent display support
- Up to 24GB DDR4 memory (8GB DDR4 soldered down)
- -40°C to +85°C operating temp., -40°C to +85°C non-operating temp.
- Support for Kontron's Embedded Security Solution (Approtect)

Kontron
www.kontron.com

Rugged COM Express Card Conforms to VITA 59

MEN Micro's CB71C is an ultra-rugged COM Express module for rail, public transportation and industry applications. It is 100% compatible with COM Express Type 6 pin-out and conforms to the VITA 59 standard, which specifies robust mechanics to ensure reliable operation even under the harshest environmental conditions.

- AMD Ryzen Embedded V1000/R1000 series
- Up to 32GB DDR4 RAM with ECC
- Up to 4 Digital Display Interfaces (DP, eDP, HDMI, DVI)
- Hardware memory encryption
- Safety-relevant supervision functions
- Support up to -40°C to +85°C Tcase, conduction cooling
- VITA 59 in process, compliant with COM Express Basic, type 6
- PICMG COM.0 COM Express version also available

MEN Micro
www.menmicro.com

Intel 9th Gen Processors Ride Type 6 COMe Board

MSC Technologies' MSC-C6B-CFLR COM Express Type 6 modules are based on the newest 9th generation Intel Core processor. The Intel two-chip solution allows highest performance in graphics and computing on a COM Express module in basic form factor.

- Intel Core i7-9850HE, Celeron G4930E or E-2276ME
- Intel UHD Graphics and chipsets QM370 or CM246
- Up to 32GB DDR4-2666 SDRAM, dual channel
- 4x SATA 6Gb/s mass storage interfaces
- 3x DisplayPort/HDMI/DVI interfaces
- Triple independent display support
- Eight PCIe x1 lanes, configurable up to x4
- Intel Rapid Storage Technology support

MSC Technologies
www.msc-technologies.eu

DATASHEET URLS:

Kontron www.kontron.com/products/come-compact/come-cwl6-e2s/come-cwl6-e2s_20190603_datasheet.pdf

MEN Micro www.menmicro.com/products/rugged-com-express/15cb71/

MSC Technologies www.msc-technologies.eu/products-solutions/products/boards/com-express-type-6/msc-c6b-cflr.html

Embedded System Essentials

Building Against Fault Injection Attacks

Cautious Coding

Fault injection are powerful attacks for bypassing security mechanisms. Rather than work on just showing the attacks, in this article Colin demonstrates how you can start to protect against them with some modest changes to your code flow.

By
Colin O'Flynn

COLUMNS

In several articles now, I've brought up the idea of fault injection (FI) attacks, and how they could be used to bypass security. I previously demonstrated this as a method of dumping a private key from a USB key (*Circuit Cellar* 342, May 2019), as well as demonstrating how you could bypass fuse bytes (*Circuit Cellar* 338, September 2018), and how electromagnetic fault injection works (*Circuit Cellar* 352, November 2019).

I also gave an overview of several fault injection attacks in my January 2018 article (*Circuit Cellar* 330). All of those have been about the offense. So, in this article, I'm going to discuss the defense—how you can help improve your code against such attacks. Check out some of those old articles for more details on how we perform FI attacks as well. With all that in mind, let's dig in!

WHAT ARE FAULT ATTACKS?

While I've covered fault attacks previously in more detail, it's worth recapping what exactly these attacks can accomplish. A fault attack is one where an attacker modifies the flow of the program, normally in order to bypass security mechanisms. These bypasses can have devastating effects. We often rely on things like fuse bytes to protect our IP programmed into a microcontroller (MCU) for example, or we rely on a signature operation to ensure that only valid code is loaded onto the MCU.

```
void write_bytes(char * data[], unsigned int datalen) {
    for(int i = 0; i < datalen; i++){
        uart_write(data[i]);
    }
}
```

LISTING 1
A simple function for sending a buffer over a serial port

Unfortunately, there isn't much you can do when the MCU features themselves are vulnerable to a FI attack. If using the LPC1114 from NXP Semiconductors that I demonstrated attacking in my May 2018 (*Circuit Cellar* 334) article (based on work by Chris Gerlinsky), you must try to rearchitect your system to work within the new security bounds. This would mean not storing any critical secrets with the flash, because you know it can be easily read by an attacker.

Luckily, not all devices have easily exploitable implementations. This means you are given a useful starting point, but it's easy to quickly shoot yourself in the foot. As two examples, let's first look at a simple output routine in **Listing 1**. This C-level code might not have an obvious exploitable defect, being just a simple loop, right? First let's take a look at **Listing 2**, which is the assembly code generated by a recent Arm GCC compiler. If you want to explore the connection between C and ASM, be sure to check out godbolt.org which is an online compiler explorer as shown in **Figure 1**.

Now, the loop ending in Listing 2 has been converted to a "branch if not equal" or `bne` instruction. This minor fact has a very significant implication for our fault injection attack: Should an attacker "skip" a single comparison during the end value of the loop, the loop will now continue to iterate until the integer value wraps around! And this type of effect is exactly what an attacker can do in practice, meaning they can suddenly dump huge sections of code. If the loop was done with a "branch if less than" instruction, the attacker would need to skip that instruction on every iteration through the loop.

This type of attack was demonstrated by Micah Scott for dumping an entire MCU firmware over USB. The link to a detailed video on this is posted on *Circuit Cellar's* article materials webpage. Other people have used the attack in a similar fashion, causing a target device to simply "read out" memory. You can see how minor changes in program flow (in this case the compiler adding a "branch if not equal") have big effects, so I wanted to take you through a few more obvious "poor design choices" that make you especially vulnerable to fault injection attacks.

FAULTY TOWERS

Now the previous example might be interesting, but it's not the most common target. In fact, the most common target is typically a signature or password check function. If you take a look at most bootloaders on the market nowadays, what you'll find is a piece of code that looks something like **Listing 3**. This type of logic is found in almost every embedded bootloader I've recently examined, so I won't single any particular vendors out (but they know who they are!).

What is the problem with this? If you skip the signature check, suddenly you are booting the unvalidated image. While the attack does require some level of physical access to perform, the typical attack vector has someone performing this attack once to dump code memory. Once the attacker has the code memory, they may be able to find other vulnerabilities or even read out sensitive (secret) keys they can then use to perform a more advanced attack.

Rather than skipping the check, another pattern is some sort of "jump to infinite loop," as shown in **Listing 4**. Again, it's not always the case that the designer intended to generate this program flow, but that the compiler may have inserted it. This infinite loop is a poor practice, since an attacker doesn't need to be particularly clever with their timing to jump out of the loop. In the example in Listing 4, we perform the same type of image validation as Listing 3. But in the program flow from Listing 4, a failed image means we go into an infinite loop that requires a system reset. But an attacker can instead send an incorrect image, and then perform a fault injection attack to skip one of the branch instructions making up the infinite loop.

```
write_bytes:
        push    {r4, r5, r6, lr}
        subs    r5, r1, #0
        beq     .L1
        sub     r4, r0, #4
        add     r5, r4, r5, lsl #2
.L3:
        ldr     r0, [r4, #4]!
        bl      uart_write
        cmp     r5, r4
        bne     .L3
.L1:
        pop     {r4, r5, r6, lr}
        bx      lr
```

LISTING 2
The resulting arm assembly code from Listing 1

Once the attacker breaks out of the loop, the code continues to execute as if a valid image was loaded. This type of code flow goes back to satellite TV smart-card days. (some of you may remember the idea of "unloopers.") It's particularly vulnerable because it doesn't require the careful timing that the code flow from Listing 3 required.

LEANING TOWERS

While the previous fault vulnerabilities might seem obvious (at least in retrospect), it's not always easy to prevent them from being attacked. Consider an attempt someone has made in **Listing 5** to protect their comparison function by adding some time jitter. The idea here being that an attacker breaking Listing 3 would be sweeping though time to find the right location where the comparison happens. By adding significant time jitter in Listing 5, it means an attacker no longer has perfect timing. This doesn't necessarily make the attack impossible, but it should make the attack much harder to replicate.

The problem is that if we again look at the assembly code in **Listing 6**, you can see the jump to the delay function could

COLUMNS

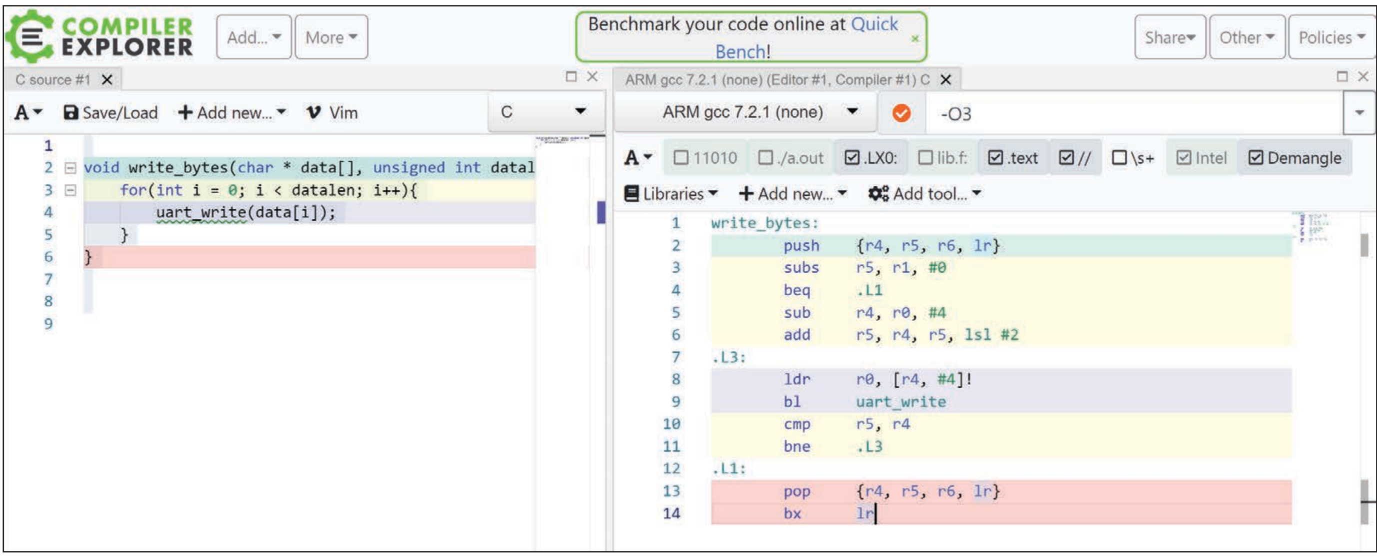


FIGURE 1
This is an example of comparing C to ASM using Godbolt.org, which will be a useful resource as you explore examples in this article.

be skipped! This might require two faults in a row, which may be reasonably practical as just requires skipping multiple instructions compared to one.

Since you likely came to this article for guidance and not more examples of incorrect code, let's move on to how we can do this correctly.

POWER TOWERS

First, we need to understand that adding fault tolerance is likely to add overhead in both code size and speed. But we can get away with some pretty minor adjustments. One example of more difficult-to-fault code is given in **Listing 7**. The major change is I've now introduced two variables. I first load the untrusted image into `test_image`, rather than directly into

```
void * boot_image;

load_image(boot_image);

if (verify_image(boot_image)) {
    jump_to_image(boot_image);
}

boot_backup_image();
```

LISTING 3
A simple bootloader that performs image verification on a received image

```
void * boot_image;

load_image(boot_image);

// verify_image() Returns -1 if
// verification fails
if (verify_image(boot_image) < 0) {
    //User must reset device to retry
    while(1);
}

jump_to_image(boot_image);
```

LISTING 4
Program flow using an infinite loop after a failed comparison

ABOUT THE AUTHOR

Colin O'Flynn (colin@oflynn.com) has been building and breaking electronic devices for many years. He is an assistant professor at Dalhousie University, and also CTO of NewAE Technology both based in Halifax, NS, Canada. Some of his work is posted on his website at www.colinoflynn.com.

Additional materials from the author are available at:
www.circuitcellar.com/article-materials

```
void * boot_image;
load_image(boot_image);
delay(random());

if (verify_image(boot_image)) {
    jump_to_image(boot_image);
}
while(1);
```

LISTING 5
Time jitter used to attempt and complicate FI attacks

```
main:
        push    {r4, lr}
        mov     r4, #0
        mov     r0, r4
        bl      load_image
        bl      random
        bl      delay
        mov     r0, r4
        bl      verify_image
        cmp     r0, r4
        bne     .L8
.L5:
        b       .L5
.L8:
        mov     r0, r4
        bl      jump_to_image
        b       .L5
```

LISTING 6
Assembly code from Listing 5 shows the delay itself can be skipped.

```
void * test_image;
void * boot_image = ERROR_HANDLER_ADDRESS;
unsigned int status = 0;

load_image(test_image);

delay(random());
status = verify_image(test_image, &boot_image)
//verify_image copies test_image to boot_image
if (status == 0xDEADF00D) {
    //Looks OK...
    delay(random());
    jump_to_image(boot_image);
} else if (status == 0xF4110911) {
    //Signature failed
    test_image = NULL;
    boot_image = NULL;
    while(1);
} else {
    //Unexpected result - fault attack??
    erase_sensitive_data();
    while(1);
}
boot_backup_image();
```

LISTING 7
Simple fault injection armored code from Listing 3

```
unsigned int verify_image(void * image, void ** boot_ptr)
{
    //We'll compare expected_hash to hash
    unsigned int expected_hash = get_known_hash();
    unsigned int hash = calculate_hash(image);

    //We also mask the value of the pointer we will jump to
    //Correctly executing code will remove these effects to
    //leave the original image pointer.
    void * possible_ptr = (void *)get_known_hash() ^ image;
    possible_ptr ^= (void *)(1 << 14);
    possible_ptr ^= (void *)(1<<15);

    //Perform multiple tests
    if (expected_hash != hash) return 0xF4110911;
    if (expected_hash == hash) possible_ptr ^= (void *)(1 << 14);
    delay(random());
    if (expected_hash == hash) possible_ptr ^= (void *)(1 << 15);
    if (expected_hash != hash) return 0xF4110911;
    delay(random());
    if (expected_hash == hash) possible_ptr ^= (void *)expected_hash;
    if (expected_hash != hash) return 0xF4110911;
    if (expected_hash == hash) *boot_ptr = possible_ptr
    if (expected_hash == hash) return 0xDEADF00D;
    return -1;
}
```

LISTING 8
Details of the verification function, which requires correct execution to result in a useful result

the final image that we'll boot. The other thing I do is push the actual assignment of boot_image into the comparison function itself, where we can do more complex operations.

Now the value of boot_image will only be set to the trusted value somewhere inside the verification function. In addition, I've made more complex return values that are less likely to be faulted. The function comparison is checked against a specific value, rather than just checked against being non-zero. Should an attacker be corrupting memory instead of skipping instructions, they will find it more difficult to corrupt memory into the specific value I'm checking. With this, I can also detect unexpected operating conditions that could be an ongoing attack. Our ability to respond will depend on the device. Such failures could in fact be innocent mistakes—such as ESD discharge or corrupted memory—but we are now making conscious decisions to deal with the detection flag.

Now I've hidden the special verify_image() function away from you, so we also need to explore that a little before I can claim I've given you a complete look. This is shown in **Listing 8**. What makes this function more difficult to glitch? First off, you'll notice comparisons are done multiple times. In this case there are several comparisons that check if the expected hash matches the calculated, and if that fails it will return a failure flag.

The other major change, is that there is no single comparison that carries the sensitive operation. You'll notice that the critical variable in this case is the possible_ptr variable. If the hash comparison is successful, this variable will get copied to the boot_image variable. But several "unmasking" steps are needed for the valid value to get loaded, including toggling several bits that will otherwise cause this to point to some invalid memory area. In theory, the call in Listing 7 to jump_to_image() with the expected image point will only occur if every comparison passes successfully in Listing 8.

Of course, this is all done from the C code level! Looking at assembly you can still identify some potential risky points. For example, what if the calls that are supposed to initialized expected_hash and hash never happen? Well, suddenly this means the entire comparison would pass! So additional guarding of the variables is needed to ensure you cannot simply skip that initial setup. But keeping fault attacks in mind is the most critical first step in designing truly secure embedded systems.

TOWER GUARD

How can you use this in practice then? I've already shown you that fault injection attacks are a serious threat to any embedded system. A careful review of your code should show you were attackers might find the most valuable targets, and you can concentrate on building fault injection resistant code around those points.

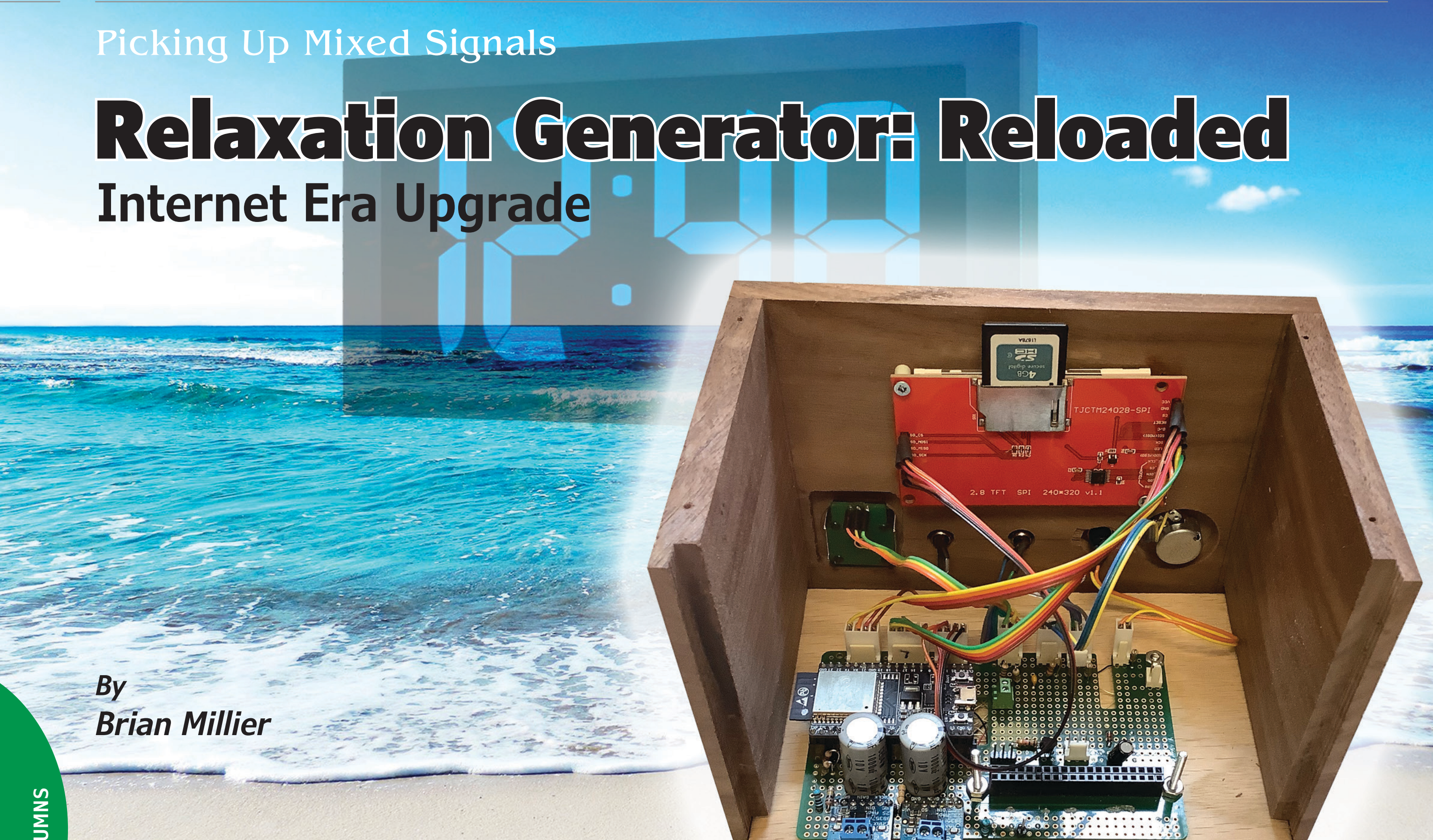
To help you out here, I've released an open-source library called ChipArmour (being Canadian I keep the "u" in Armour) that uses some of these best practices. You can either use the library as a reference for building fault-injection resistant code, or directly integrate it into your firmware project. This library is released under a permissive Apache license, so you can use this in both your own open-source and commercial projects.

My future columns will explore ChipArmour in more detail. This is still in an early beta, so you may not find a complete build available when you are reading this column. But I wanted to first bring you through the specifics of how fault injection attacks can be applied to a simple codebase, and how you can reduce the vulnerability of your existing code to fault injection attacks with some small modifications.

Picking Up Mixed Signals

Relaxation Generator: Reloaded

Internet Era Upgrade



By
Brian Millier

Some years ago, Brian wrote an article for *Circuit Cellar* about his project that generates relaxing sounds—ocean waves, rainfall and such—and inculpating a digital clock to shut off the sounds. At the time, he built it with only Atmel 8-bit AVR MCUs and support chips. In this article, Brian describes his more modern version of the project, this time built with an Espressif ESP32 MCU to provide Internet connectivity.

About 10 years ago, I published a *Circuit Cellar* article about a project I had designed that could generate relaxing sounds, such as ocean waves, brooks and rainfall. I ran this device at night for help falling asleep, and to mask out random outdoor noises that would wake up our dogs, whose barking would then wake us up. Incorporated in the project was a digital clock with an alarm feature that shut off those sounds.

The design for that project had to be more hardware-intensive 10 years back. At the time, I was using only Atmel 8-bit AVR microcontrollers (MCUs), and I had to choose a model that was close to top-of-the-line to get the functionality I needed (Atmel is now part of Microchip Technology). I also needed five other support chips to complete the design. I re-designed the project once—about 5 years ago—when I started using Arm MCUs. More recently, I decided to build a more modern version, with an Espressif ESP32 MCU to provide Internet connectivity. In this article, I describe this newest version of my project.

REAL-TIME-CLOCK CHOICES

Because alarm clock functions were important to this project, I needed a real-time-clock (RTC) circuit of some sort to handle the time-keeping. Because power outages sometimes occur where I live, I wanted an RTC that would maintain the time through a power outage. In my first model, that function was handled by Maxim Integrated's DS1307, and, in a later version, a Philips PCF8563. Both versions used a 3.3V coin cell as the battery backup. The design of the earlier models was such that powering the entire unit from a battery was not practical. Power supplies of 10V, 5V and 3.3V would be needed. Therefore, when a power failure occurred, the "Wave" sound would stop. If you have used one of these relaxation devices, you know—as we found—that once the sound stops, you quickly wake up. For the earlier versions, this was a shortcoming. At least the clock never needed to be reset, because the RTC chip was backed up by the coin cell.

This time around, I decided that I wanted the entire unit to be capable of running from a battery

for 12 or more hours. That eliminated the need for a discrete RTC chip, since the ESP32 MCU can maintain the correct time completely in software—as long as it's powered up. The newest version needs no manual setting of the clock, because the ESP32 connects to my home Wi-Fi router, and gets its time setting from an Internet-based Network Time Protocol (NTP) server.

The main reason I was able to power the whole project from a battery for an extended period partially stems from the choice of an extremely efficient digital audio amplifier module for this version. The earlier versions used a Class-B linear power amplifier (NXP Semiconductors' TDA1517), which produced excellent quality sound but needed a 10V power supply and drew a significant amount of current.

SOUND FILE DATA STORAGE

One aspect of the project that didn't change over the 10 years was how the sound files were stored. I wanted to have several different sounds available. These sounds are played repetitively in a "loop," but you want each of them to have some variety over time, so they should be at least a few minutes long. It turns out that the sounds of brooks and ocean waves involve a significant amount of high audio frequencies, so I settled on the 16-bit/44kHz sampling rate (CD standard). Furthermore, I produce these sounds in stereo, with one speaker on a bedside table on each side of the bed. This gives a much more immersive sound.

It turns out that no low-cost, serial flash EEPROM devices are available that can handle the amount of data that these several files would contain. However, inexpensive SD cards are readily available. Even the lowest-capacity SD cards now available have much more storage capacity than is needed for even 25 such sound files. I chose an LCD display that contained an SD card socket, eliminating the cost and wiring of a separate SD card socket.

Although I have built devices that reproduced the popular, highly compressed MP3 file format, I did not consider this format here. That's because it would require either an external MP3 decoder chip such as the VS1033, or a significant amount of processing by the ESP32 MCU. The ESP32 is capable of MP3 decoding, and software libraries are available. However, I didn't see any advantage in using a compressed sound file format, given the huge amount of storage available on even the smallest SD card. The ESP32 has other tasks to perform in the project, and there was always a chance it would not be able to handle everything in real time, with no glitches in the sound output.

I chose the standard Microsoft .WAV file format, because it is well documented and easy to handle in software. Another advantage is that you can find "relaxing" nature sound files readily on the Internet, and these files are generally in the .WAV format. The .WAV format contains one or more sections of metadata in various "headers," prior to the large block of data containing the actual waveform data. Although these headers contain useful information—such as the song name, data rate and the number of bits resolution—I don't try to parse out this information from the header sections (called "chunks").

The project is designed for a sample rate of 44,100Hz, 16-bit stereo data, and that is the format that the .WAV file(s) must be in for proper operation. Therefore, all I must look for in the file is the word "data." Once I find that, the next 4 bytes make up a 32-bit number defining the length of the actual waveform data. I use this value to determine when I have reached the end of the waveform data. Immediately following the 4 file-length bytes are the actual data, and that is where I start reading the waveform data.

Figure 1 shows a hex dump of the beginning of an actual .WAV file that I use, with the "data" bytes circled in green. Although the bytes making up the string "data" are in the expected order, the following 4 bytes defining

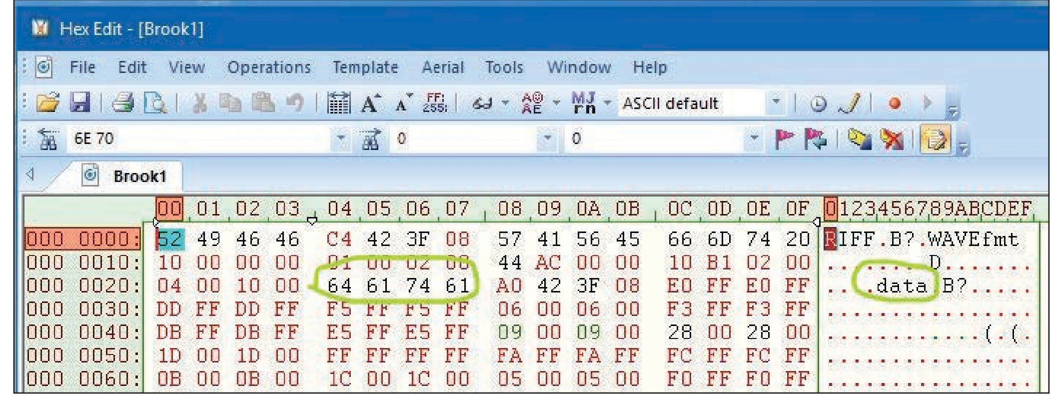


FIGURE 1
A hex dump of the beginning of a .WAV file. The start of the data "chunk" is marked by the ASCII string "data," which I've circled in green.

COLUMNS

the file length, are in the big-Endian format, so you have to read them "backwards." That is, the 0xA0423F08 value shown after "data" in Figure 1 equals 138,363,552 bytes. This file happened to be an hour long. In practice, one could use files that were only a few minutes long, as they are looped, and there is no "dead" (muted) time interval between the end of the file and when it starts back at the beginning.

THE CLOCK DISPLAY

One aspect of my earlier designs that wasn't ideal was the clock display. Initially, I used a common 20×2-character LCD display with an LED backlight. It was easy to dim the LED backlight, so that it was not so bright as to disturb sleeping. However, as with all LCD character displays, the font was small and hard to read at any distance. I designed my own larger font using four adjacent character positions, so it was useable.

For the next version, I used an Adafruit 4-digit LED display module. I chose it because it contains its own controller chip and is interfaced via I²C. The Arm MCU module that I was using (a Teensy 3.2) did not have a lot of spare GPIO pins, so the two-wire I²C interface was essential. The controller on this module can set 16 different LED brightness levels (by adjusting the LED current). However, I found that even the lowest brightness level seemed too bright for my liking at night. Even placing a colored filter in front of the LED module didn't dim it enough.

For my latest version, I chose a common and inexpensive 2.8" color TFT display. An LCD display produces no actual light of its own, but merely filters/blocks the light emitted from its LED backlight. I control that backlight using a PWM (pulse width modulation) pin on the ESP32, so users have complete control over how dim they want the display to be. The software adjusts the backlight brightness, depending on whether it's day or night. The controller library for this display contains the ability to use many different fonts/sizes, and I chose one that displays 0.5"-high characters, which are easily readable even when you're half asleep!

One consideration that I initially overlooked, when choosing a graphic TFT LCD display, was the amount of time it would take to update the clock display. The TFT display is interfaced via SPI, and the ESP32 can handle high SPI data rates (40MHz). However, there is more to it than that. To simultaneously produce the relaxation sounds, the SD card (also an SPI device) must be accessed at a high enough rate to provide 176,400 data bytes per second. The SD card interface cannot handle the 40MHz SPI rate, however.

The waveform data must be transferred via the I²S bus to the two DACs that provide the 44.1KHz/16-bit stereo sound output. The DACs themselves have no internal buffers, so they must be fed data at a steady rate of 176,400 bytes/s, to produce "glitch-free" sound output. Therefore, the time it takes to update the TFT display must not interfere with the steady data flow needed for the sound output.

I found it interesting to note that on my 10-year-old version of this project, I was able to accomplish this audio streaming with an 8-bit ATMega644 MCU clocked at 20MHz, using a single interrupt service routine and some hand-coded assembly language. The 32 bit ESP32 MCU runs at 240MHz and contains two cores. Its I²S library routine uses DMA transfers. Even with all this MCU horsepower and DMA, it was tricky to accomplish the TFT clock display update, without introducing any glitches into the audio playback. More on this later in the "Software" section.

SD_Mode status	External Resistor	Selected Channel
HIGH	0Ωto V_{IN}	Left
Pull-up through R_{SMALL}	470kΩ to V_{IN}	Right
Pull-up through R_{LARGE}	floating	(Left + Right)/2
LOW	0Ωto GND	Shut down

TABLE 1
The four different modes available on the Adafruit breakout board

GAIN_SLOT configuration	Gain (dB)
Connected to GND via 100kΩ resistor	15
Connected directly to GND	12
Unconnected	9
Connected to V_{DD}	6
Connected to V_{DD} via 100kΩ resister	3

TABLE 2
GAIN_SLOT configurations on the Adafruit breakout board

THE DACs: MAXIM MAX98357

In my original version of this project 10 years ago, the Atmel ATmega644 MCU that I used was among the fastest 8-bit MCUs available. But it didn't contain an I²S port. Neither did most general-purpose MCUs of the day. Therefore, I used a common MCP4822 SPI 2-channel 12-bit DAC, and followed it with a TDA1517 linear stereo power amplifier chip.

This time around I used two MAX98357 devices from Maxim Integrated. The MAX98357 contains an I²S 16-bit DAC and a Class D audio amplifier, capable of putting out 3.2W of power using only a 5V power source. I used two of these for stereo. This choice helped to reduce the overall power

consumption to a level where four AA batteries could be used for backup power lasting for 12 hours or more.

These devices come in either a very tiny WLP (wafer-level packaging) package or a tiny TQFN (thin quad flat no leads) package. There is no way I can personally solder such a small device to a PCB by hand. Adafruit comes to the rescue again, by selling a breakout module containing a MAX98357 device. The price of the Adafruit module is very reasonable considering it would cost about half that price for the MAX98357 IC, alone.

The MAX98357 requires three of the standard I²S signals: BCLK, LRCLK and DATA—but does not require the higher frequency MCLK signal normally needed by many other audio codecs, DACs and other devices. This is important. Although the ESP32 can produce the high-frequency MCLK signal, it can only route that signal to GPIO0, which may not be available in some project designs.

If you feed the same three I²S signals to both MAX98357 chips, how does each device know if it is the left or the right channel? This is handled in a clever way on these devices. There is a single analog input pin (SD_MODE) that determines in which of four modes it will run. In the case of the Adafruit module, there is a 1MΩ pull-up resistor connected to the SD_MODE pin and the MAX98357, itself, has an internal 100kΩ pull-down resistor. The four different modes available on the Adafruit breakout board can be achieved as shown in **Table 1**.

The MAX98357 devices use BTL (bridge-tied load) outputs—that is, the two output pins are differentially-driven, and neither one should be connected to ground. This rules out the use of headphones with the MAX98357, since headphones generally have Left, Right and Common wires. Driving two separate speakers is fine, though. The last feature of the MAX98357 is the adjustable Gain pin. If you assume that the I²S digital signal being fed into the MAX98357 is at full scale (0dBV), the output signal level is: Output Signal Level (dBV) = 2.1dB + selected Amplifier Gain (dB).

The Amplifier Gain is determined by the configuration of the Gain Slot pin, labeled Gain on the Adafruit breakout board (**Table 2**). Regardless of the digital input signal and amplifier gain, the maximum voltage that the MAX98357 can put out is limited by the 5V suggested maximum V_{DD} limit. Because of the BTL output configuration, the maximum signal output is 2 × 5V or $10V_{PP}$ (minus small voltage drops from the internal MOSFET output drivers). According to the MAX98357 specs, the maximum power output with a 4Ω speaker is 3.2W, which corresponds to a peak-to-peak output signal level of 8.9V.

Using a full-scale I²S digital input and the maximum gain of 15dB, the output signal level would be 2.1 + 15 = 17.2dBV. This corresponds to $7.24V_{RMS}$ or 20.5V peak-to-peak, which is more than double the maximum output voltage, so a lot of distortion would occur. Clearly, the 12dB and 15dB gains are meant to be used only when the I²S digital input signals are much less than the digital, full-scale values.

DIGITAL VOLUME CONTROL

I needed to have a volume control in the unit. However, since the signal chain is digital all the way to the speakers, the only way to accomplish this is in the software. The 16-bit digital waveform values coming from the SD card's .WAV file must be divided by some constant, which is derived from the volume control's setting. The 10kΩ volume control in the project is fed from a 2.5V reference IC through a 15kΩ resistor, which places 1V across it. The ESP32's internal ADC has a voltage reference of 1.0V. Using the ADC to measure the pot position, the wiper's voltage will span the entire ADC input range. I use the 8-bit ADC value to determine the constant mentioned above.

I must admit I didn't look too closely at the calculations shown in the previous MAX98357 DAC section, before I decided to go with the MAX98357 digital amplifier modules for this project. I had them on hand and had used them for an earlier project. In that project, I was pleasantly surprised that each MAX98357

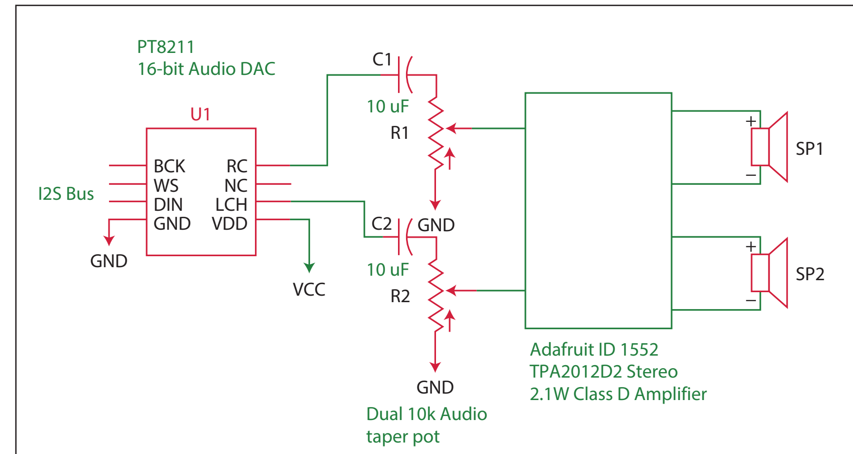


FIGURE 2
This block diagram shows what would have been a better audio output circuit than the one I had chosen.

ABOUT THE AUTHOR

Brian Millier runs Computer Interface Consultants. He was an instrumentation engineer in the Department of Chemistry at Dalhousie University (Halifax, NS, Canada) for 29 years.

could drive an older hi-fi loudspeaker cabinet with a 12" woofer (and tweeter), adequately filling a 250ft² room.

For this project, I was using only two small speaker enclosures with 5" woofers. More importantly, the sound levels needed would be much lower, since you are trying to sleep while the unit is operating. Therefore, I wired the Gain_Slot pins for the minimum 3dB gain setting. At this low gain setting, the signal output level (with an F.S. digital input) would be 5.1dBV (1.8V_{RMS}) or about 0.8W (double for two channels). It turned out that significantly less power per speaker was needed for comfortable audio levels.

The digital volume control is working with 16-bit integer waveforms. I reduce the default amplitude of the 16-bit waveform by multiplying it by some value in the range of 1 to 255, based upon the setting of the volume pot. Then I divide this value 256, by arithmetically shifting the number left eight times. For the amount of attenuation that I found was needed to produce a reasonable sound level at night, this works fine and doesn't reduce the resolution of the audio waveform enough to be objectionable.

In hindsight, I realize that I could have made a better design choice for audio output. Given the small amount of audio power actually needed, I could have used a circuit like the one shown in **Figure 2**. That would have eliminated the need for software control of the volume, which would have eased some of the software timing constraints I had to handle. I have numerous PT8211 DACs on hand. I had to buy ten at about $1 each. However, they are not readily available through normal USA distributers. Also, the TP2012D2 Class D amplifier could have been replaced by two Texas Instruments (TI) LM386 linear power amplifier ICs. Even with a V_{CC} of only 5V, they would have put out enough audio power, and not used a whole lot of current.

THE CIRCUIT DIAGRAM

Figure 3 is the overall circuit diagram. The ESP32 chip and supporting components/antenna are mounted on what Espressif calls the ESP32 DevKitC module. Espressif first produced these, and still sells an updated version. Not all the DevKitC modules use the same pin layout as what I show in the diagram. My module has male pins mounted

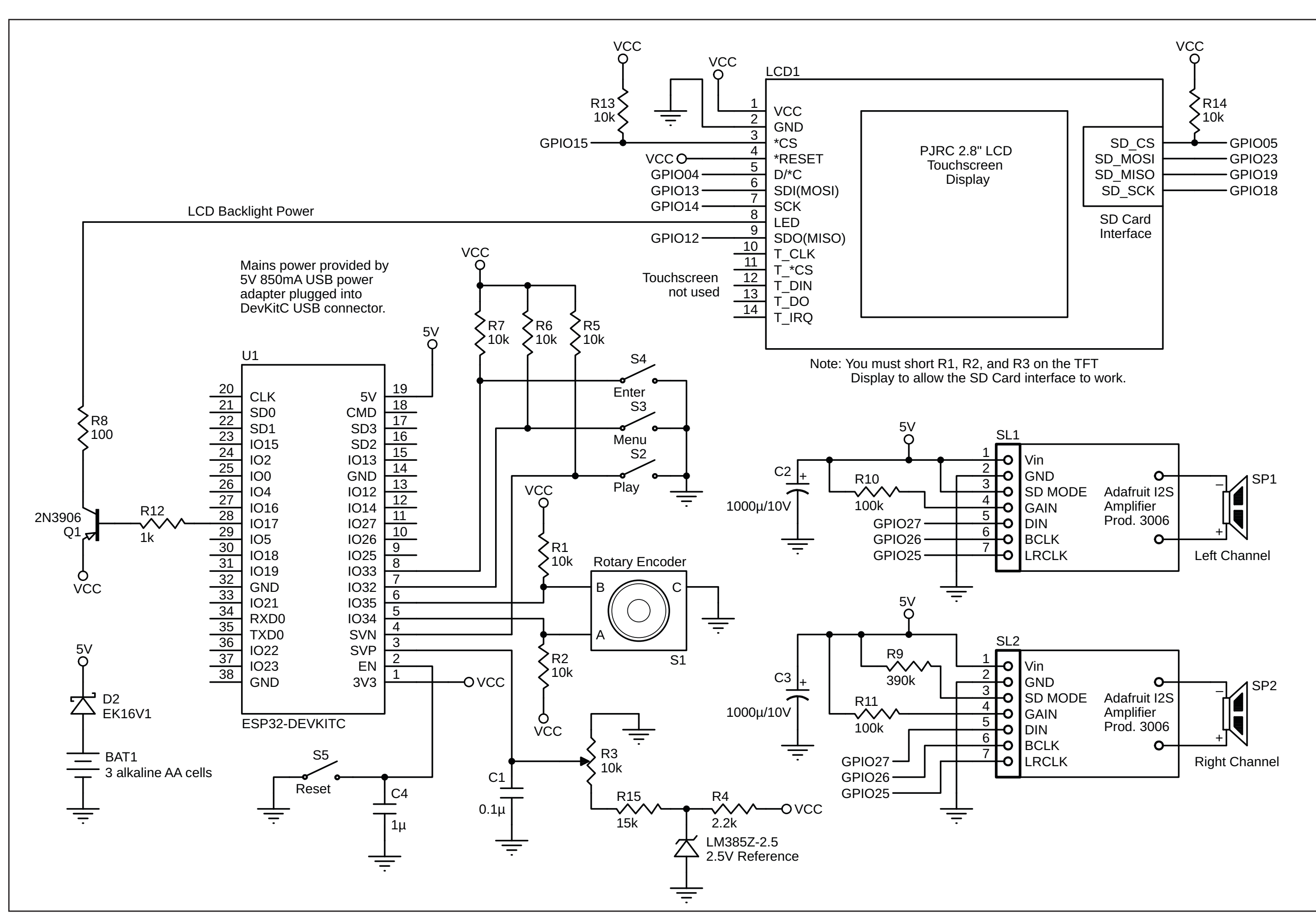


FIGURE 3

Schematic of the complete unit. Note that the PJRC color TFT display needs to have three resistors jumpered out, in order for the SD card socket to work reliably.

on the bottom of the PC board, as shown in **Figure 4**.

For some reason, the newer Espressif DevKitC modules have female headers mounted on the top of the PC board. That would mean, for example, that I couldn't swap in the newer model into my project because the pins would all be flipped 180 degrees. I don't know how you are expected to use the Boot and EN buttons on these newer boards, because they would be covered up by the module when it was plugged in.

The ESP32 DevKitC is programmed via its micro USB port, using the built-in serial bootloader. I do my ESP32 software development using the Arduino IDE, loaded with the ESP32 boards package. With the Arduino IDE, downloading ESP32 program code to the DevKitC is simple: the "normal" ESP32 requirement of depressing the Boot button, while toggling the EN button on/off to download code is unnecessary. This is handled by the DTR and RTS handshake signals coming from the Silicon Labs' 2102 USB/serial bridge device on-board the DevKitC module. The Arduino ESP32 downloading tool toggles the DTR/RTS lines properly, whereas other ESP32 downloading applications may or may not do this.

One issue with some of the ESP32 DevKitC modules I have used concerns the power-up reset. During project development, I kept the DevKitC plugged into my PC's USB port for power and for programming purposes. Connected this way, the 2102 USB/serial bridge will assert the DTR/RTS signals, so that the ESP32 will reset properly and start program execution as soon as the DevKitC board is enumerated by the PC as a valid USB com port device. However, the ESP32 would not execute a normal power-up reset when I tried to power the project using any of the following setups:

1) A USB power adapter plugged into the DevKitC USB socket
2) A battery pack consisting of three AA cells, feeding the VIN pin
3) A 3.7V LiPo battery feeding the V_{IN} pin

The DevKitC module has a 3.3V LDO regulator on board to power the ESP32, so a battery supply to the Vin pin will work properly if the battery is greater than 3.3V (and ideally not much more than 6V). When using either of the two different battery sources, the ESP32's V_{IN} supply voltage should have risen to full value immediately, except for some delay charging the two 1,000µF capacitors (C2 and C3), which act as reservoirs for the two MAX98357 amplifier modules. In the case of the USB power adapter, the 5V would have risen somewhat slower than either of the batteries would have.

In all the aforementioned three cases, it appeared that the power to the ESP32 was not coming up to specs quickly enough for a proper ESP32 reset to occur. I temporarily removed the two 1,000µF capacitors, but that didn't help. After consulting ESP32 forums, I ran across this issue in several posts. I eventually solved it by adding a 1µF capacitor (C4) to the ESP32's EN (reset) line. Note that the V_{IN} pin is actually labeled "5V" on the DevKitC, though it needn't be a regulated 5V, as noted earlier.

SPI INTERFACE

I mentioned before that it was tricky to stream the audio data from the SD card to the I^2S DACs, while also updating the TFT display without introducing audio glitches. Both the TFT display and the SD card interface use an SPI interface. The TFT display can handle SPI transfers at the 40MHz maximum SPI clock rate that the ESP32 can put out. Even at this high rate, I measured the display update time at 9.6ms, and that only involved updating the current time using five large-font characters.

Many fancy fonts are available with this library, but they require you to "erase" the screen area "under" the characters when refreshing the display, or you will just add the new character's pixels on top of the old character, resulting in an unreadable display. Therefore, I chose the most primitive "block" font, since it did not need erasing and thus updated more quickly.

The SD card's SPI interface can't handle the 40MHz SPI rate. In fact, the Arduino SD card library runs at an SPI rate of only 4MHz. Most of the current Arduino libraries for SPI peripherals use what is called a

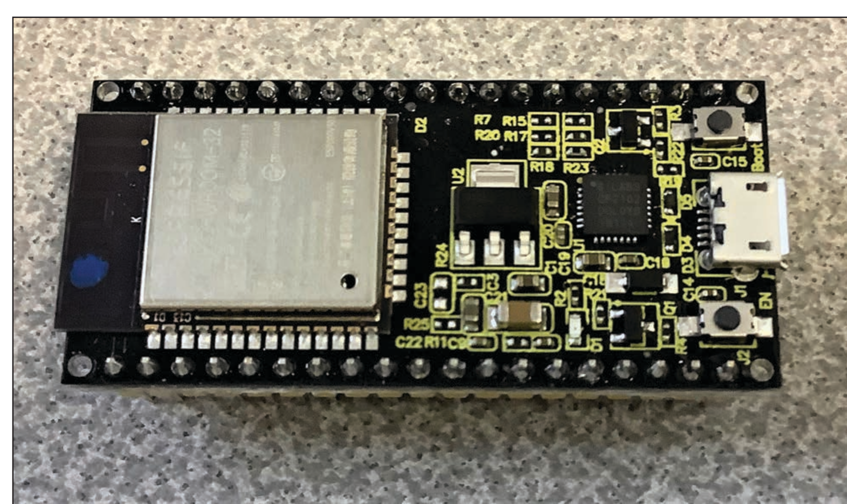


FIGURE 4
Photo of the ESP32 DevKitC. The module I used has its male pins on the bottom of the PCB. Newer versions by Espressif have female headers mounted on the top of the board.

COLUMNS

"transactional" approach. That is, any library functions that directly perform SPI transfers will set up the SPI port for the proper SPI mode and clock rate parameters (as configured for that peripheral), prior to sending each SPI message. Therefore, if there is more than one device sharing the SPI bus, the SPI port will be properly configured for each peripheral in advance as it is accessed. This was a big advance for Arduino SPI libraries, in such cases. But it does slow things down a bit.

For this project, I decided to use both SPI ports available on the ESP32—a third is dedicated to the DevKitC's flash memory device. The TFT display is driven by the ESP32's HSPI port and the SD card is driven by the VSPI port, which is the default SPI port used by most ESP32 libraries that use SPI. I didn't delve deeply into either the SD card or the TFT display libraries enough to know if using both ports was any faster than using only one SPI port. But early in my coding, I was experiencing audio glitches until I got the code optimized, so it was worth doing it this way.

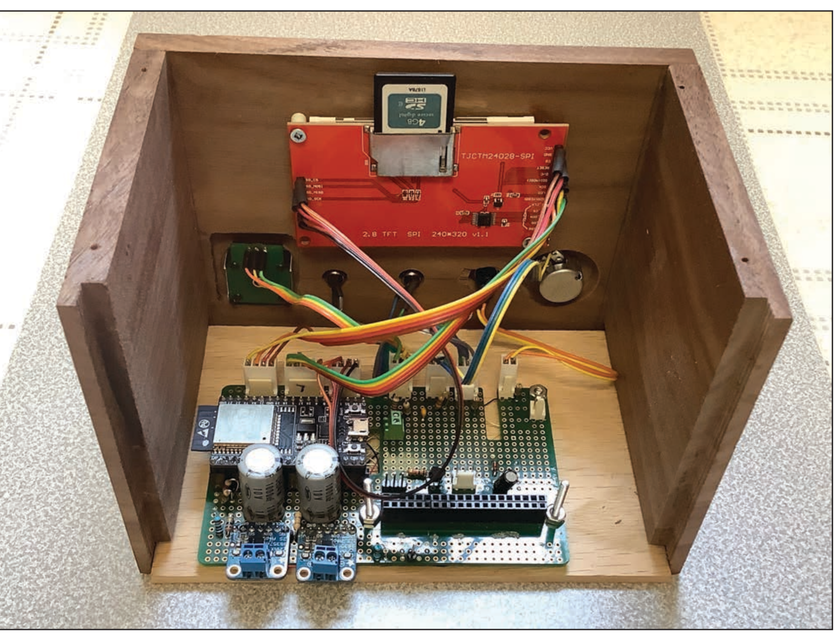


FIGURE 5
Photo of the unit from the back in its cabinet. The four AA cells are not shown, because they are mounted on the back panel.

Both MAX98357 DACs use the I²S port. This is a synchronous serial protocol that requires 4 bytes of audio data to be sent to the DAC at the chosen sample rate (44,100Hz). This data transfer must be a steady flow. There is no buffer inside the DAC to handle data, if it were to be sent in a burst mode. Luckily, Espressif has written a DMA-driven I²S library that handles this task. Since it is DMA-driven, it acts in the background, and other program code, such as fetching the next sector of data from the SD card, can operate concurrently.

TFT DISPLAY

The display I used is a 2.8" TFT touchscreen display with a resolution of 320 × 240 pixels. As just mentioned, it uses an SPI interface that can handle the high speeds put out by the ESP32's HSPI port. I generally get these displays from PJRC.com, and they work very well. I recently got one of them from another source, and while it worked, it was too dim for normal use. I decided to use that one in this project, as I need a dim display for nighttime use anyway.

While this display includes a touchscreen, I didn't use that feature. I know that the touchscreen and the associated `XPT2046_touchscreen` Teensy library from PJRC work well. I find using touch on such a small display to be awkward, so I decided to use three switches and a rotary encoder for the user interface. This TFT display includes a standard-sized SD card socket. I had seen posts on forums claiming that the SD card interface on this display didn't work. It turns out that there are three resistors (R1,2,3) on the board that must be jumpered (shorted out). With that taken care of, the SD card interface worked fine, using the ESP32's VSPI port.

To dim the display at night, I used a PWM output from the ESP32 to feed Q1, a PNP transistor. This provided a PWM-controlled current to the display's LED backlight. The ESP32 contains a very sophisticated "LEDC" controller. It can generate up to 16 PWM signals on user-defined GPIO pins, completely in hardware. In my article "Exploring the ESP32's Peripheral Blocks" in *Circuit Cellar* 332 (March 2018), I discussed this peripheral in detail, along with a few other unique ESP32 peripherals. Today there are high-level library routines available to configure these

For detailed article references and additional resources go to:
www.circuitcellar.com/article-materials

RESOURCES

Adafruit | www.adafruit.com

Cadence Design Systems | www.cadence.com

Espressif Systems | www.espressif.com

Maxim Integrated | www.maximintegrated.com

Microchip Technology | www.microchip.com

NXP Semiconductors | www.nxp.com

PJRC | www.pjrc.com

Silicon Labs | www.silabs.com

Texas Instruments | www.ti.com

U-blox | www.u-blox.com

COLUMNS

peripherals, but when I wrote that article, they weren't available, so I wrote my own routines.

POWER CHOICES

The project is normally powered by a USB power adapter capable of at least 500mA, which is plugged directly into the micro USB socket on the DevKitC. For battery backup, I decided to use three AA cells instead of a LiPo battery. Because power failures are infrequent, I assumed the battery backup would be used only sporadically. The shelf life of alkaline batteries approaches 10 years, so they wouldn't have to be checked often. Three fresh AA cells will put out 4.8V. I placed a Schottky diode in series with the positive battery supply lead to prevent current from the 5V USB power module from entering the battery chain.

The project uses about 100mA when it is not playing any sound, and about 150mA when it is playing sound. This varies somewhat depending upon what level of dimming you apply to the TFT display. During the day, it contacts an NTP server once every hour to synchronize the time. This is a bit of overkill, and could be reduced to once per day without affecting anything. During synchronization, the current will increase, with short spikes of around 250mA for up to 15 seconds while the ESP32's Wi-Fi circuitry is operating. The AA alkaline cells are rated around 2,400mA-hours so they should last for 12 or more hours. **Figure 5** is a rear-view photo of the project in its case. The AA cells are not visible, because they are mounted on the back cover.

The protoboard I used here is a specialty board meant to mount on top of a Raspberry Pi. I had previously mounted the two MAX98357 DAC/amplifier modules on this board for a Raspberry Pi project that I had decided not to pursue. The finished unit is shown in **Figure 6**.

SOFTWARE

When I switched to using the Arduino IDE from Bascom-AVR for my AVR projects, it was mainly because of the wealth of libraries available from thousands of Arduino enthusiasts. It turned out to be a wise choice, since this IDE has been expanded to handle many different Arm MCUs, of which I use Teensy 3.x and 4.0. It also handles the ESP8266/ESP32—which are not even Arm-based, but rather Tensilica Xtensa. (Tensilica is part of Cadence Design Systems.) Currently, I am using Visual Micro, an add-on to Microsoft's Visual Studio. This VM/VS combination acts as a "wrapper" around the Arduino C++ toolchain, and provides a much better programming environment.

FIGURE 6
Shown here is the completed unit mounted in a small enclosure I made from walnut.

For this project, several critical libraries were needed to handle the task, all of which would have been difficult to develop on one's own:

1) The SD card library to read the sound data files from the SD card
2) The TFT graphic library for the display
3) The I2S DMA-driven library to feed the DACs
4) The NTP library to set/synchronize the ESP32's software-driven RTC with network time

The Arduino SD card library was originally written for AVR MCUs, but when you add the ESP32 board package to the Arduino IDE, you get a custom SD card library written by Espressif. The TFT touchscreen display uses an ILI9341 controller chip. Adafruit originally wrote the `Adafruit_ILI9341` library for the AVR family, and it has been customized more recently to handle Teensy, ESP8266/ESP32 MCUs. It calls the `Adafruit_GFX` library for its graphics features. Important Note: My program uses the ESP32's HSPI port for the TFT display's SPI access, whereas the Adafruit ILI9341 library uses the VSPI port by default. This change is handled in my code as follows:

```
SPIClass SPI2(HSPI);

And in setup()

SPI2.begin();
tft.begin(0, SPI2);
```

The above code works fine with version 1.1.0 of the Adafruit_ILI9341 library, but it won't compile with later versions, because they have changed something. You must use the Arduino "Sketch - > Include Library - > Manage Libraries" function to load this older version of the library, if that is not the one you are currently using. The I²S DMA-driven library is written by Espressif. They use a certain style for their libraries, which differs from many other Arduino libraries. The Espressif libraries operate under the free RTOS operating system, and the I²S DMA-driven library needs the following included files:

```
#include "driver/i2s.h"
#include "freertos/queue.h"
```

The I²S port is configured by filling up the following two structures:

```
i2s_config_t i2s_config = {
  .mode = (i2s_mode_t)(I2S_MODE_MASTER | I2S_MODE_TX),
  .sample_rate = 44100,
  .bits_per_sample = (i2s_bits_per_sample_t) 16,  //I2S_BITS_PER_SAMPLE_16BIT
  .channel_format =   I2S_CHANNEL_FMT_RIGHT_LEFT,
 .communication_format = (i2s_comm_format_t) (I2S_COMM_FORMAT_I2S | I2S_COMM_FORMAT_I2S_MSB),
  .intr_alloc_flags = ESP_INTR_FLAG_LEVEL1, // high interrupt priority
  .dma_buf_count = 8,
  .dma_buf_len = 64,   //Interrupt level 1
  .use_apll = (int) 1
};

i2s_pin_config_t pin_config = {
  .bck_io_num = 26, //this is BCK pin
  .ws_io_num = 25, // this is LRCK pin
  .data_out_num = 27, // this is DATA output pin
  .data_in_num = -1   //Not used
};
```

The I²S port is started up as follows:

```
i2s_driver_install((i2s_port_t)i2s_num, &i2s_config, 0, NULL);
i2s_set_pin((i2s_port_t)i2s_num, &pin_config);
```

Since the I²S port is DMA-driven, when you want the sound to stop, it is not enough to just stop filling the DMA buffers. If that's all you do, you'll get a constant buzz coming from the speakers. You must add the following line to de-activate the DMA driver:

```
i2s_driver_uninstall((i2s_port_t)i2s_num);
```

As far as the NTP time synchronization is concerned, I had already done a few earlier ESP8266/ESP32 projects that needed NTP time synchronization. I didn't use any library, instead adding all the code needed to do the initial UDP request and handle the NTP reply. I set the timeserver string variable to time.nist.gov URL and let the ESP32 resolve the IP# by using:

```
WiFi.hostByName(timeServer, timeServerIP);
```

I thought I'd be clever this time and use a higher-level NTP library that I found on GitHub, which seemed simple to use. Basically, you just call this routine and pass it your wireless router's SSID/Password, and it does everything necessary to synchronize the ESP32's software RTC. During the many hours spent developing/programming this project, I found that this NTP library routine didn't always work, and ultimately it failed to work at all.

Examining the library code, I saw that it used a fixed IP# for the NTP server. From past experience, I knew that the IP#s of these servers change from time to time, and the method described in the last paragraph was more reliable. So, I reverted to using my own, "non-library" code, and it has worked well so far. I will say that you do have to wait a bit after sending an NTP request for the response to come back (if it's going to), and you also must allow for a number of retries if you want to be sure of getting a valid NTP synchronization.

The user interface is quite simple. The first time that the ESP32 is powered up after the project code has been loaded, it will check out the first two bytes of EEPROM for the "0x55, 0xAA" signature. Since the user hasn't configured the project

yet, these two EEPROM bytes will instead be in the default erased state. The program will then call the configuration routine where it will ask for:

1) The desired Alarm time
2) The sound file # (from a list of sound file names on the display)
3) The local time offset from UTC. I don't specifically handle Daylight Saving Time changeover in software, so you must change this offset twice a year on the day that the time changes.

These parameters will then be saved to EEPROM, where they will remain intact if the unit is powered down by removing both AC power and the battery.

The user interface consists of the following:

1) Three push buttons
2) Menu: To select the configuration menus listed above
3) Enter: to enter the value of the parameter being modified
4) Play: To start/stop the playing of the relaxation sound. Once started, this will continue to play until the alarm time is reached, or the user hits Play again.
5) A rotary encoder to adjust parameter values
6) A potentiometer to adjust volume

An SD card must be inserted into the SD card socket containing at least one sound file in the .WAV format. When using SD cards in an MCU-based project, it is always good practice to format the SD card using the "SDFormatter" PC application.

CONCLUSIONS

I've now built three versions of this device over 10+ years, all of which worked well. Since I use it every night, it is one of my projects that, in addition to being interesting to build/program, I also use repeatedly. That's my justification for spending the additional time designing the newer models. I haven't mentioned that one of my original goals in building this newest version was to incorporate a GPS module. This module would:

1) Provide an accurate time (in place of the external, Web-based NTP server)
2) Provide a "local" NTP server that could be used by several other IoT devices I've built for my home, all of which have an ongoing need for the correct time/date. Currently they use the same "external" Web-based NTP server that this project does.

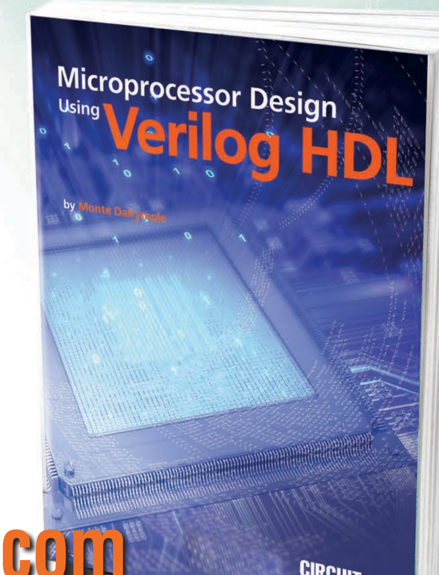
I was stymied by this part of the project. An older GPS module that I had in my "spare" parts bin turned out to be dead. I ordered a GPS board from Amazon that contained the common U-blox Neo-6M module and a tiny antenna with 3" of coax cable. While I was able to see a lot of NMEA messages spewing out of it, it only rarely would get the proper "fix" on enough satellites to provide the time, never mind my location. I gave up trying out the unit in a window with a good "view" of the sky and took the whole thing outdoors. Even then it worked horribly. So, I returned it for a refund, and decided to abandon that part of the project. I had planned on placing this project a few feet from a large window, and I doubt that the inexpensive GPS modules that I was looking at would have worked properly.

This failed experiment makes me suspicious when I see TV shows where someone hides a "GPS" tracker underneath a car. What kind of great antenna must those trackers use?

COLUMNS

Okay, maybe not easy, but certainly less complicated. Monte Dalrymple has taken his years of experience designing embedded architecture and microprocessors and compiled his knowledge into one comprehensive guide to processor design in the real world.

Monte demonstrates how Verilog hardware description language (HDL) enables you to depict, simulate, and synthesize an electronic design so you can reduce your workload and increase productivity.



The Consummate Engineer

Semiconductor Fundamentals (Part 5)

More on FETs

George continues his article series delving into the fundamentals of semiconductors. In Part 5, he expands his discussion of field effect transistors or FETs. He examines different types of JFETs and MOSFETs, looking at aspects including gate architecture and drain-source I-V characteristics.

By
George Novacek

Last month we started our discussion of field effect transistors (FETs). Now, let's expand on the topic. The FET was patented by Julius Edgar Lilienfeld in 1926, preceding the invention of the bipolar junction transistor (BJT) by some two decades. But the FET was never constructed, as the necessary technology wasn't available at that time.

Let's start with junction, or JFET structure, as depicted in **Figure 1** with its accompanying schematic symbols. JFETs come in two flavors: N-channel and P-channel. The control electrode called gate is a P-N junction forming a diode which must be reverse-biased to achieve the FET's signature high input impedance. The reverse-biased gate inhibits the movement of electrons or holes, based on whether the channel is of N or P respectively. At zero bias, as shown in **Figure 2**, the maximum drain-to-source current flows. And because the negative gate bias decreases the drain current, the JFETs are called depletion-mode devices.

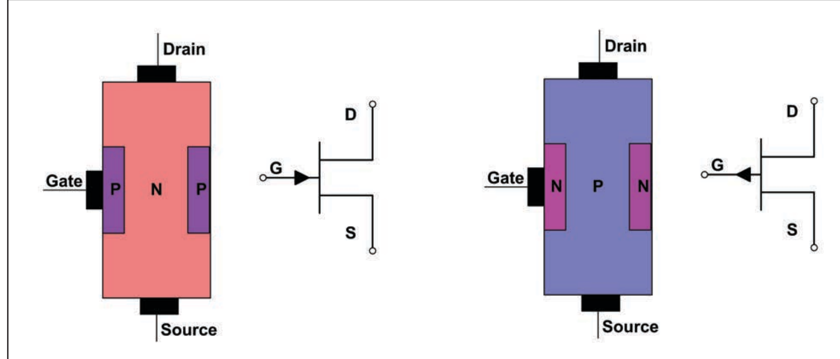


FIGURE 1
Structure of N-channel JFET and schematic symbols of N and P channel

Analogous to BJTs, FET amplifiers can also be created in three basic configurations: Common source, common gate and common drain—the last also known as a source follower. Figure 2 plots the drain-to-source current I_D versus drain-to-source voltage V_{DS} with gate-to-source voltage V_{GS} as a parameter. At some negative V_{GS} called pinch-off voltage V_P the drain current will be zero. In the ohmic region the JFET acts as a voltage-controlled resistor:

$$R_{DS} = \frac{\Delta V_{DS}}{\Delta I_D} = \frac{1}{g_m} \tag{1}$$

where R_{DS} is the channel resistance and g_m is the FET's transconductance gain.

JFETs' dice are smaller than BJTs'. So, the common source JFETs are often used in the long-tailed pair configuration in front-end stages of monolithic op amps. This topology has very high input impedance and good voltage gain. Common gate topology can be seen as the second stage of a cascode amplifier, also

analogous to the BJT version with the same advantages and disadvantages. The source follower is frequently an integral part of high resistance sensors—such as the pyroelectric ones—because it matches their high input resistance to a low resistance output.

In saturation the I_{DSS} current shows very little dependence on the drain-to-source voltage. This makes the design of constant current sinks and sources easy (**Figure 3**). Resistor R adjusts the magnitude of the sink current which, for R = 0 is the maximum saturation current I_{DSS}. To set a lower current the resistor R value is increased. A P-channel JFET operated at opposite polarities forms a constant current source.

JFET APPLICATIONS

As shown in Figure 1, JFET is a symmetrical device, so the drain and source terminals are interchangeable. This property makes JFET useful as an analog switch or a voltage-controlled resistor. Applications include volume control, voltage-controlled oscillators, modulators and wherever a variable resistance is needed. Because of their high input impedance and small geometry, FETs also find their use in low noise, high frequency circuits up to about 30GHz.

Figure 4 is a basic N-channel, common source JFET amplifier. When setting up its DC operating point you must remember to keep the gate electrode negatively biased with respect to source. Here the gate is connected to the ground potential V_G = 0 and the source electrode's potential is raised by drain current I_D through resistor R_S to V_S, just as we used to bias triodes. With the working I_D established from the JFET I-V diagram, you compute resistor values R_S and R_D to set V_D to approximately $(V_{DD} - V_S)/2$. Of course, you need to know the JFET characteristics—unfortunately, specification sheets I have checked give you very little data—just enough for designing an analog switch, but nothing more. You need to do some measurements of your own.

In the Figure 4 amplifier example I used J110 JFET with R_G= 1MΩ, R_S = 470Ω and R_D = 1kΩ. The DC operating point of this amplifier was V_S = 2.3V, V_D = 7.15V, V_G = 0V and V_{DD} = 12V. To obtain a useful AC gain I bypassed R_S with a 100µF capacitor, which resulted in the gain of 23dB (AV ≈14), - 3dB flat from 26Hz to 21MHz.

As mentioned earlier, JFETs come as N-channel and P-channel. The N-channel is doped with donor impurities and, therefore, the current through the channel is negative in the form of electrons. The P-channel is doped with acceptor impurities and, consequently, the current through the channel is positive in the form of holes. Because electrons have higher mobility than holes, N-channel JFETs exhibit greater channel conductivity (lower resistance) than their P-channel counterparts. The N-channel JFET is more efficient and, therefore, while available, P-channel JFETs are not as frequently used.

MOSFETs

The next step is to contemplate the metal-oxide semiconductor field effect transistor or MOSFET. Just like bipolar transistors and JFETs, MOSFETs come as N and P types. Additionally, each type can be an enhancement or depletion, so that makes our MOSFET types. Most digital ICs today—including microprocessors, microcontrollers,

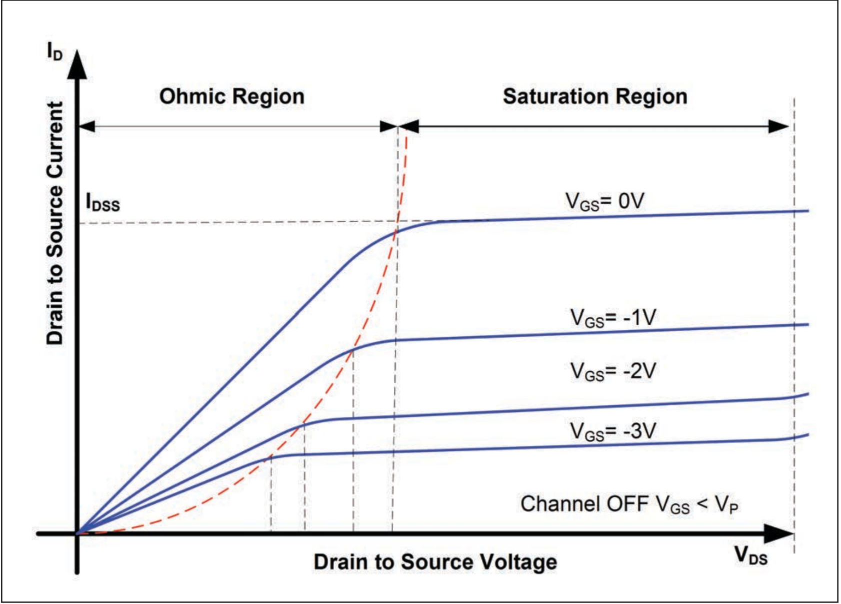


FIGURE 2
JFET Drain I-V characteristics with V_{GS} a parameter

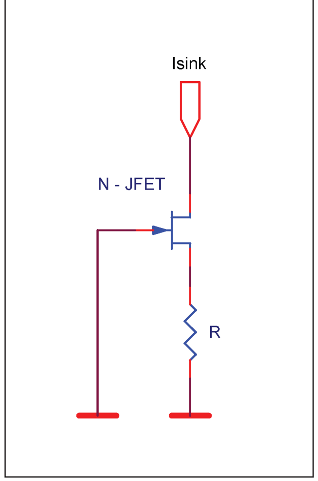


FIGURE 3
N-channel JFET constant current sink

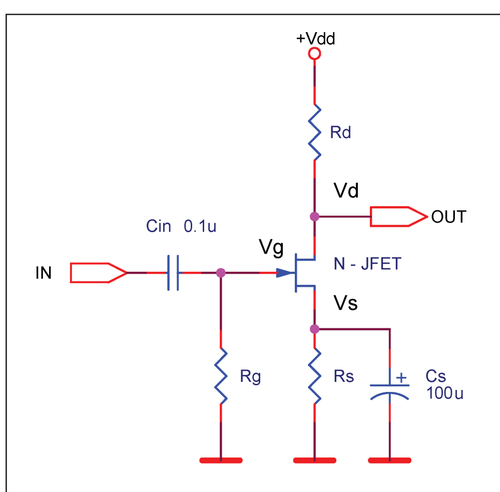


FIGURE 4
A common source low frequency amplifier

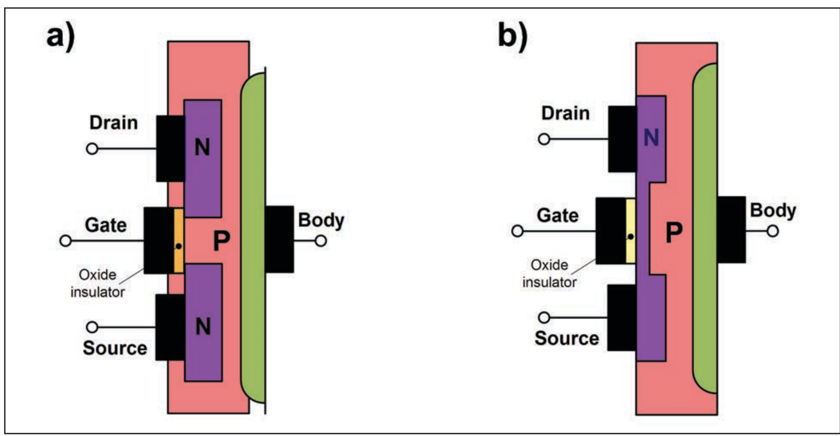


FIGURE 5
Cross-section of N-channel enhancement (a) and depletion (b) type MOSFET

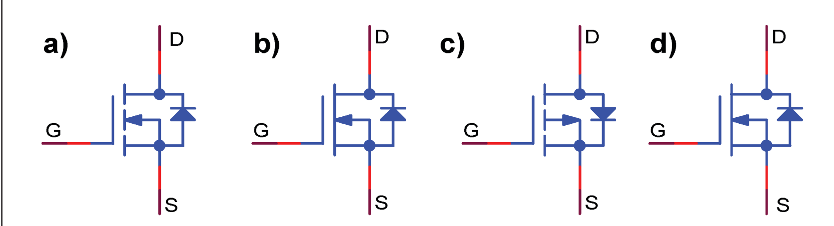


FIGURE 6
Shown here are the symbols of enhancement (a) and depletion (b) type N-channel MOSFETs and enhancement (c) and depletion (d) type P-channel MOSFETs. No bulk terminals are present.

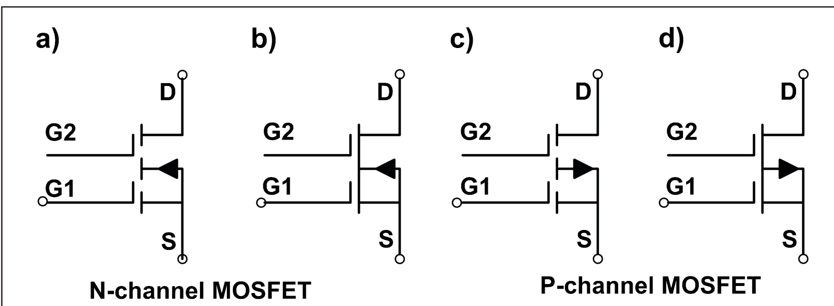


FIGURE 7
Here are the symbols of dual gate enhancement (a) and depletion (b) type N-channel MOSFETs and dual gate enhancement (c) and depletion (d) type P-channel MOSFETs.

ABOUT THE AUTHOR

George Novacek was a retired president of an aerospace company. He was a professional engineer with degrees in Automation and Cybernetics. George's dissertation project was a design of a portable ECG (electrocardiograph) with wireless interface. George contributed articles to *Circuit Cellar* since 1999, penning more than 120 articles over the years. George passed away in January 2019, but we're grateful to be able to share with you this, and a couple more articles he left with us to be published.

storage devices and so on—are made using MOSFET technology. In the '70s, purely PMOS or NMOS ICs had been fabricated, but today complementary pairs of the N- and P-channel transistors let us build CMOS devices with high speed and low power consumption.

The main difference between the JFET and the MOSFET is that the MOSFET gate is isolated from the body of the semiconductor by an oxide insulator. Therefore, an extremely high input resistance to the tune of 1,012Ω is obtained. I used such a MOSFET to interface with an ionization chamber that had a cross-current around 1-2nA. Because the gate could be easily destroyed by static electricity discharge, many MOSFETs have an internal diode protection, somewhat degrading the high input resistance.

Figure 5a illustrates the N-channel enhancement MOSFET structure, **Figure 5b** shows the N-channel depletion mode structure. Notice that MOSFETs have a fourth electrode called body, bulk or substrate. It is generally connected to the source potential. Many MOSFETs have the connection done internally and no lead outside the enclosure. Also notice that the substrate-to-drain and substrate-to-source junctions form intrinsic diodes. With the bulk usually connected to the substrate, the substrate-to-source diode is shorted. But the substrate-to-drain diode is the reason why the MOSFET drain and source are not interchangeable. The diodes are shown in the MOSFET symbols (**Figure 6**). The P-channel MOSFET's structure looks the same, just the polarities are reversed. The N-channel enhancement MOSFET, in my experience, is the most prevalent for switching and digital applications.

By now MOSFETs have replaced BJTs in many applications, including communications reaching up to the many gigahertz. Dual gate MOSFETs were developed especially for applications as oscillators, mixers, multipliers, amplifiers and so forth in RF, VHF, UHF, microwave and higher frequency ranges. Both gates affect the operation of the device, which could be viewed as two MOSFETs in series. Their respective symbols are shown in **Figure 7**. Notice the intrinsic diode is not always shown in the MOSFET symbol.

A dual gate MOSFET can form a cascode amplifier overcoming the Miller effect as discussed previously. The Miller effect relates to the impedance between the output and the input, but at high frequencies capacitance is the predominant factor, potentially leading to instability. Biasing Gate 2 (also called the drain gate) at a constant potential, well bypassed to the ground, eliminates the capacitive coupling and thus the Miller effect.

The I-V characteristic of depletion MOSFETs

is similar to that of the JFETs in Figure 2. Enhancement MOSFETs need some minimum gate to source voltage to begin to conduct as seen in **Figure 8**. Design of an amplifier with enhanced mode MOSFETs is along the same lines as described earlier for JFETs. Just the gate biasing is different. MOSFETs intended for logic switching applications guarantee minimum R_{DS}—in other words, the drain-to-source resistance, often in milliohms, at the gate voltages less than 5V.

BUILDING BLOCKS

Some readers may think discussing discrete components unnecessary. Thanks to the availability of many inexpensive integrated circuits (ICs) and system building blocks, circuit design with discrete components is becoming almost an arcane art. But, all that said, transistors are the building blocks of ICs—devices that many make a living designing. And even if IC design is not in your future, understanding their underlying principles can only help with product designs and troubleshooting.

We'll complete the series next month with a look at power MOSFETs and some nearly exotic components with multiple P-N junctions.

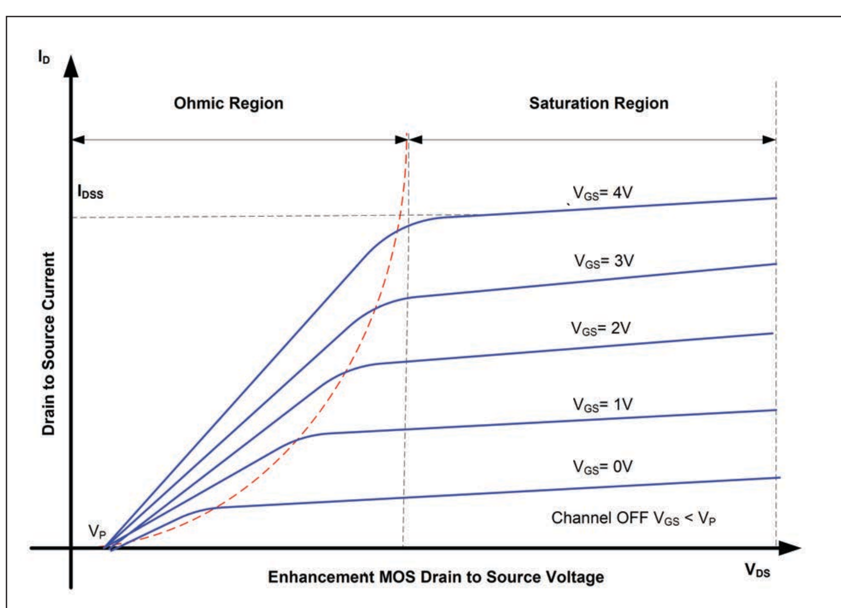
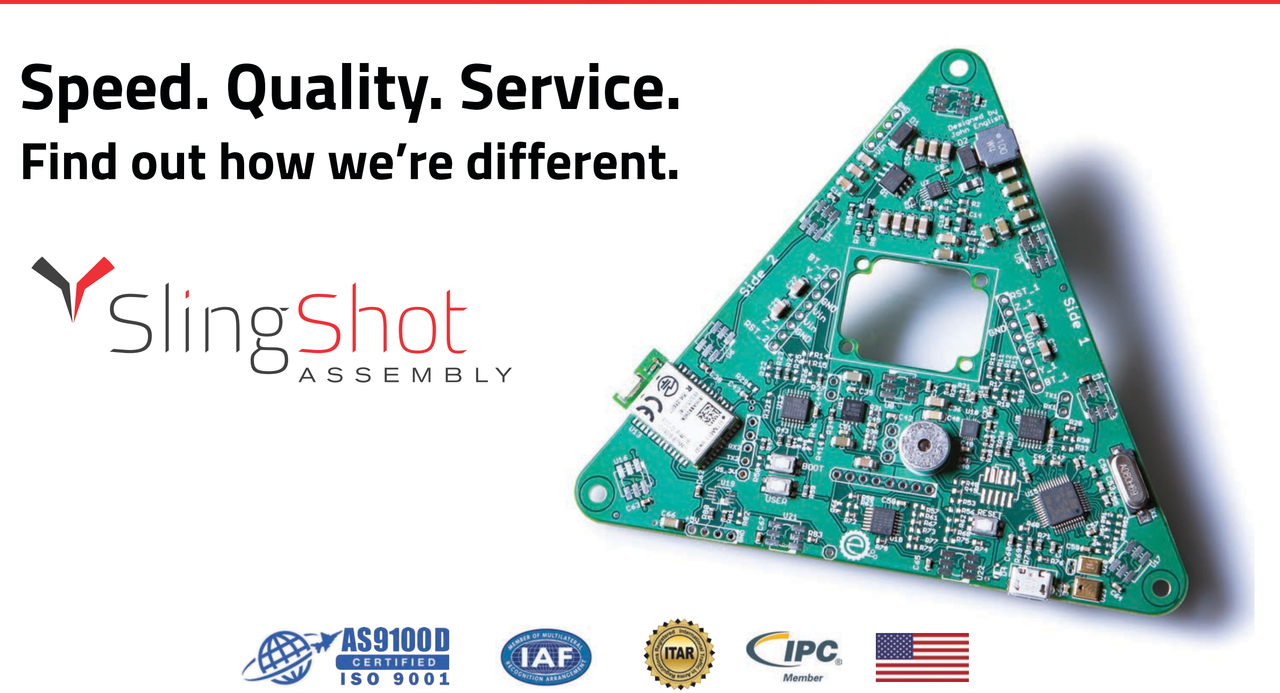


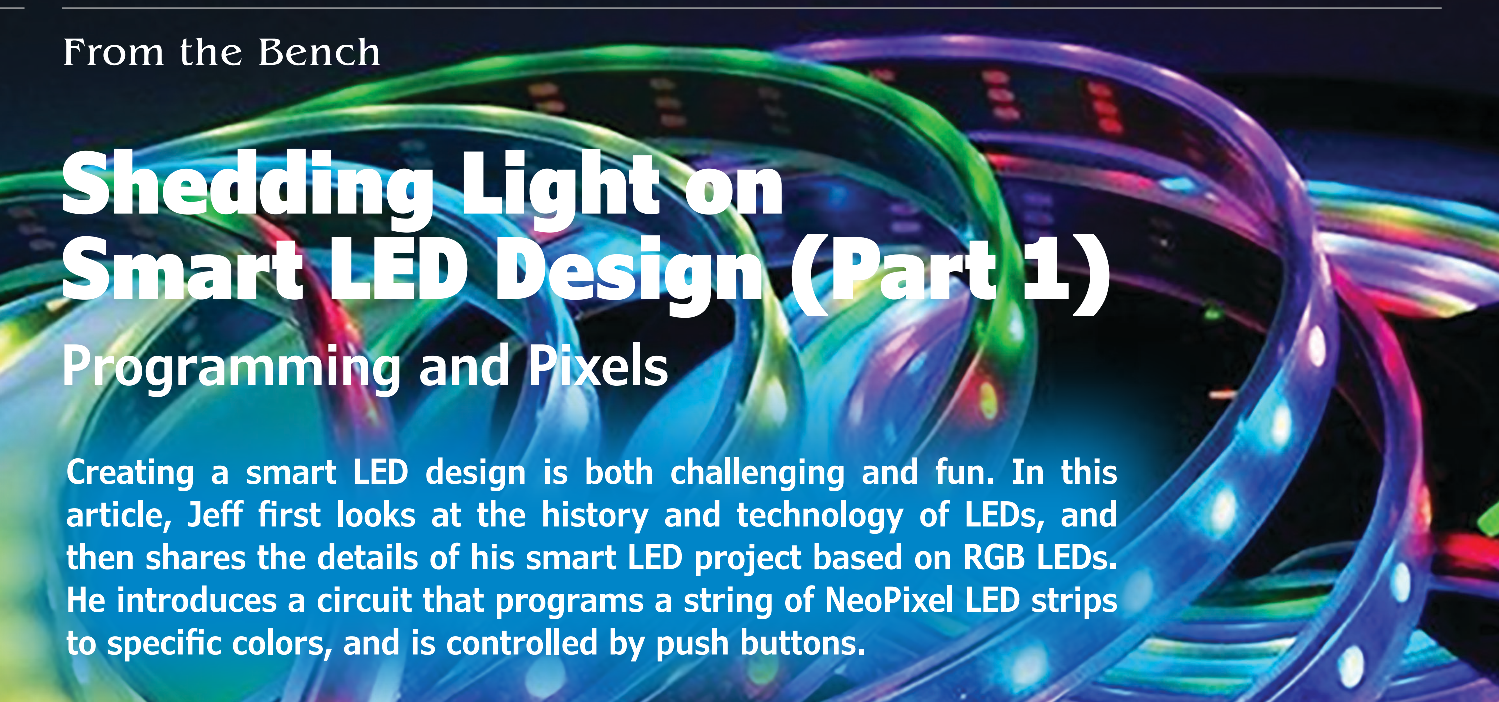
FIGURE 8
Drain–source I-V characteristics of an enhancement type N-channel MOSFET

For detailed article references and additional resources go to: www.circuitcellar.com/article-materials

COLUMNS



From the Bench



Shedding Light on Smart LED Design (Part 1)

Programming and Pixels

Creating a smart LED design is both challenging and fun. In this article, Jeff first looks at the history and technology of LEDs, and then shares the details of his smart LED project based on RGB LEDs. He introduces a circuit that programs a string of NeoPixel LED strips to specific colors, and is controlled by push buttons.

By
Jeff Bachiochi

COLUMNS

It's hard to imagine how the Earth could be illuminated by just the stars and the planetary bodies in the night sky. This is like a concert hall or arena being lit just by cigarette lighters (for you older readers) or cell phones (for you younger readers). Two thousand years ago, our ancestors had no concept of the universe or what was producing the points of light seen in night sky. If the moon happened to appear, the amount of light cast upon the Earth increased dramatically, depending on its phase. Unaware that this was reflected light from the sun, our ancestors knew its light far exceeded that of the stars and gave the moon its magic quality. Even with a full moon, this low level of light causes a problem for our eyes.

Typical LED Characteristics			
Semiconductor material	Wavelength	Color	V_F @ 20mA
GaAs	850-940nm	Infra-Red	1.2V
GaAsP	630-660nm	Red	1.8V
GaAsP	605-620nm	Amber	2.0V
GaAsP:N	585-595nm	Yellow	2.2V
AlGaP	550-570nm	Green	3.5V
SiC	430-505nm	Blue	3.6V
GaInN	450nm	White	4.0V

FIGURE 1
Listed here are some semiconductor materials used to produce an LED's P-N junction, along with the wavelengths (colors) emitted and the typical band-gap voltages required [1].

Our eye's retina is made up of rod and cone cells. Rod cells are sensitive to low light levels in the green/blue area of the spectrum. These cells are unable to differentiate colors, so objects seen in low light appear to be black or white (shades of gray). The sun brings forth a blinding light (compared to the night). It was easy to be in awe of this mighty illumination that divides the day from the night. No wonder our ancestors thought of these as gods, lording over the heavens.

With sufficient light from the sun, three types of cone cells in the retina take over. Each type of cone cell responds to a specific light frequency range. The combination of the light levels received by each of these cone cells determines the color of the light being received in that area. So, the colors we see are, in fact, made up of the combination of only three cell outputs—just like the pixels of an HDTV create the illusion of any color using only three different colored LEDs (red, green and blue or RGB). This month we'll look at the smart LED, made using RGB LEDs.

LEDs

Electroluminescence, a material's ability to emit light in response to the presence of an electric current, was observe in the early 20th century. Practical LEDs weren't available until the 1960s, and even then, only red LEDs were available. The color produced by an LED is based on the materials used, and it would be another 10 years before other colors were produced with adequate output. The band gap requirements—energy required to cross a junction—of each material is different, so we have different voltage requirements for each type of LED (**Figure 1**) [1].

Today, we have tri-color LEDs available in both through-hole and surface-mount (SMT) configurations (**Figure 2**). If you look closely at the SMT device in Figure 2, you may suspect there are more than three discrete LEDs in this SMT package. While discrete LEDs are available in SMT, this picture is of a NeoPixel from Adafruit, the smart LED I'll be using for this project. The NeoPixel is the combination of an addressable IC (WS2811 or similar) and RGB LEDs. This IC handles the intensity of each of the three LEDs, based on three 8-bit values shifted into the device. It has just four connections—power, ground, and serial in and out. Let's take a closer look on how to work with it.

Anyone who has worked with graphics of any kind is probably familiar with the 24-bit digital signature used for describing pixel color. The 24-bits are actually three 8-bit bytes, with one byte for each of the RGB colors. This means that you have 8 bits of control over the intensity of each of the three colors, where 0 is fully OFF and 255 is fully ON. This 24-bit value is sent in an RGB sequence, MSB first. Since there is no external clock necessary, the WS8211 requires the serial data to conform to a special format for each bit.

For a bit=0 the T_{ON} time must be between 150-450ns, and the T_{OFF} time between 750ns and 1,050ns. For a bit=1 the T_{ON} time must be between 450-750ns and the T_{OFF} time between 450-750ns. The total time for a cycle (T_{ON} + T_{OFF}) is between 650ns and 1,650ns. A pause in excess of 50µs ends a sequence. Refer to **Figure 3** for this, and note that the NeoPixel is designed to allow multiple devices to be daisy-chained S_{IN} to S_{OUT}.

You must send out 24-bits for each NeoPixel connected in series. Note that data flow through each device is 1 bit out for each bit in. So, the first 24 bits you send out will end up in the last device in the chain. When a pause in serial data occurs (greater than 50µs), all devices latch onto the bits in its shift register. This latched data will be used to set three PWM outputs to drive the LEDs. Note here that there is a 500ns delay between a bit in and a bit out of each device. The latching of each device's data is therefore not synchronized. This will be perceptible only for very long strings of LEDs, because these delays add up.

Undoubtedly, you have noticed that the timing is fairly fast. This is good, because it allows you to update an entire string of LEDs quickly. But, it's also bad, because this shifting will most likely require blocking other execution while active. If a bit time is about 1µs, then it will take 24µs for each NeoPixel and 2.4ms for a string of 100. That's a long time to block any other routine.

Now that I've stated the facts as in the WS2811 sheet, let the truth be told. If you plan to work with NeoPixels, you'll want to read Josh Levine's WordPress blog [2] on the subject of bit timing. It seems that as long as you use the proper ON times, OFF timing can be much more relaxed, and this greatly improves the ability of your code to work with other interrupting sources.

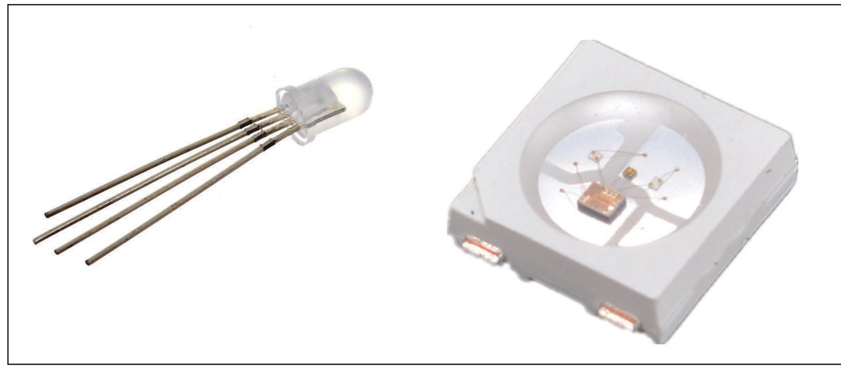


FIGURE 2
Today we can purchase RGB LEDs in both through-hole and SMT packages.

CODING

I decided to give these constraints a test try by coding only the ON times in an interrupt. The extra code in the interrupt assures a minimum OFF time, and additional interruptions in between bits can extend this OFF time. As long as this doesn't exceed the reset time, 50µs, we should be good. This month's project uses a PIC16F1847 from Microchip Technology, which is an 18-pin flash microcontroller (MCU), and this circuit will be part of a larger project. This circuit consists of eight (switch) inputs along, with one output for driving a string of nine NeoPixels and a few miscellaneous I/Os for communications. The schematics are shown in **Figure 4** and **Figure 5**.

This MCU will run at 32MHz with its internal oscillator, giving an execution speed of 8MHz or 125ns. Each bit or ON time is coded as a "bsf" (bit set instruction), some delay and a "bcf" (bit clear instruction). The delay is different for a "0" and a "1" bit, and consists of NOPs (no operation instruction). Note the timing in each of the two Timer1 routines in the NeoPixel interrupt.

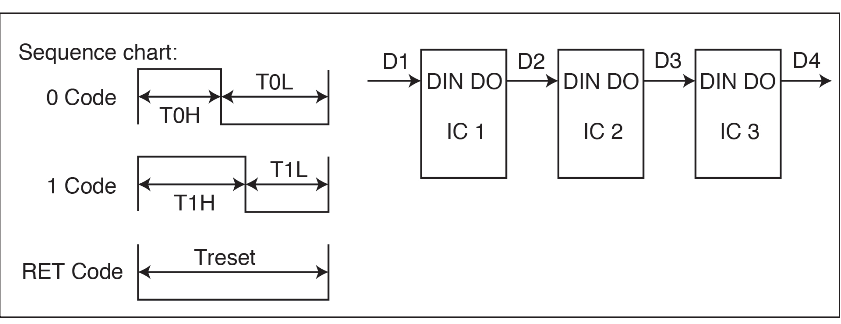


FIGURE 3
On the left are the timing specs for communicating with the WS2811. The ON time determines the bit's value. Data must be sent continuously, until the last daisy-chained device receives the first 24 bits sent. An extended OFF time latches the present data into all devices.

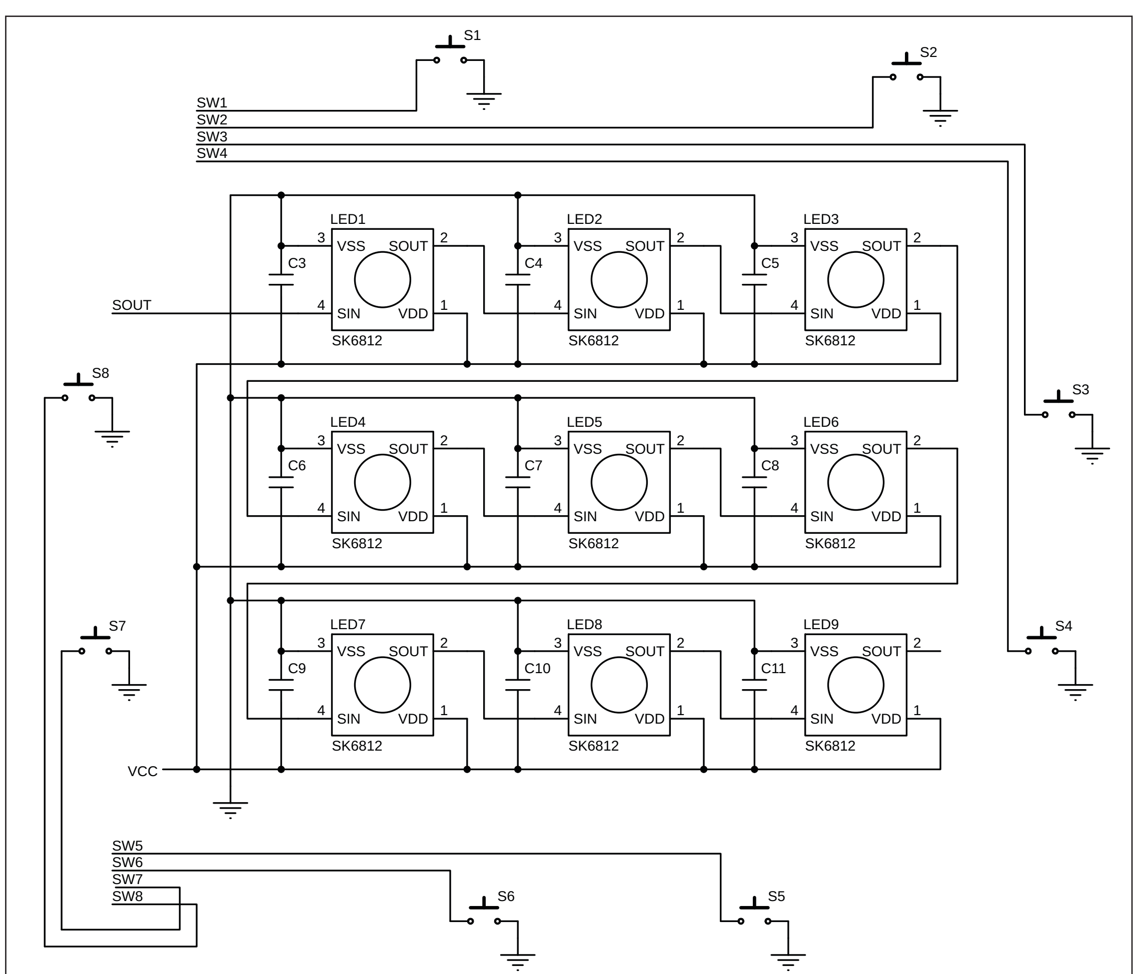


FIGURE 4
This schematic shows the eight input switches and nine serial NeoPixels for this month's project PCB.

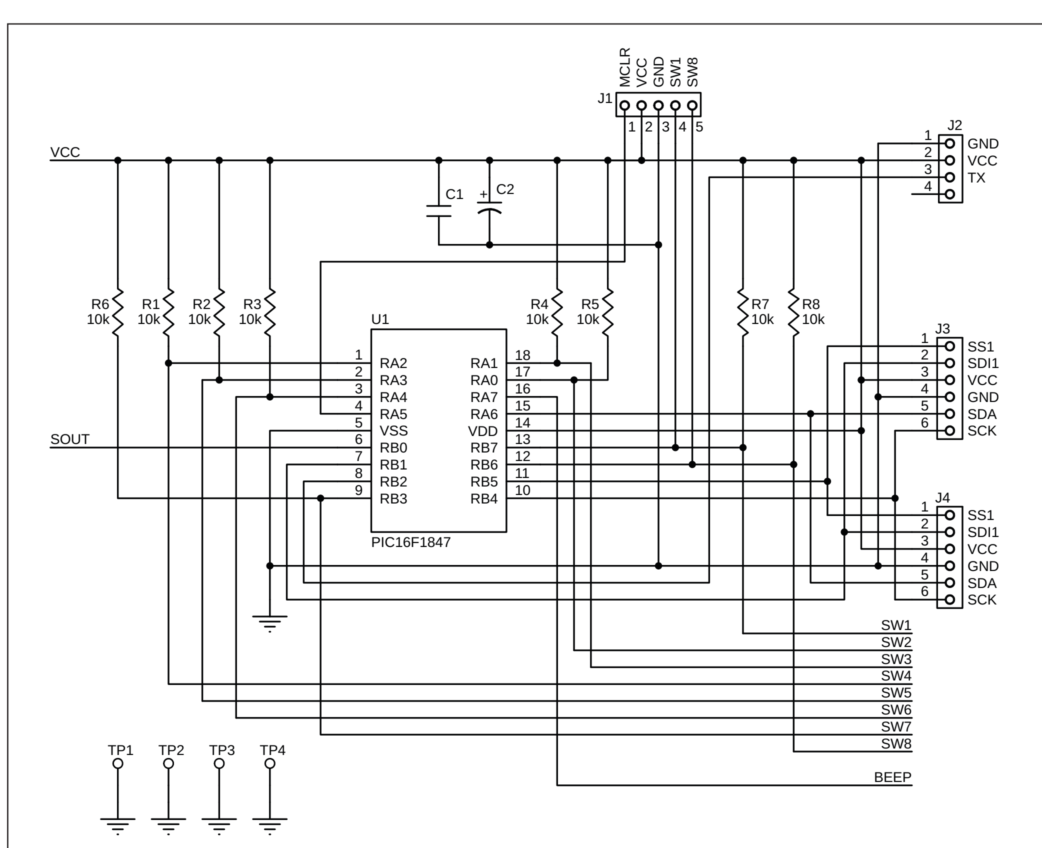


FIGURE 5
Schematic of the PIC16F1847 MCU and communication, programming and debugging connectors. All parts are SMT parts except the switches and pin headers.

COLUMNS

These adhere to the max and min ON times for NeoPixel bit. While Timer 1 had a 30ns resolution, it was easier to code two fixed timing routines than to load the timer and let the timer count a more exact timing. This would end up being a totally blocking interrupt, if not for the relaxed OFF timing. So, I'm just using the `TIMER1` interrupt without actually using its timing ability! The remaining code must fetch a byte of data to send, strip off the data bit in question (MSB to LSB), and set/delay/clear the output bit, before leaving the routine. A check after each bit is sent disables the `TIMER1` interrupt once all bits have been transmitted.

The NeoPixel data—3 bytes for each of the nine NeoPixel LEDs—are stored in the proper sequence as required by the NeoPixel data format. This makes fetching each byte easy, using an indirect register move. Picking off the appropriate bit of each byte is done by rotating the byte through the carry. The ON time code is based on the carry, as shown in **Listing 1**.

A BIT (24 BITS) ABOUT COLOR

The website rgbcolorcode.com [3] offers a graphic example of how RGB values change as they are mixed to produce a required color. You'll note that a color's code is different, depending on whether you are adding light—as in this project (RGB)—or applying pigments, as with printing (CMYK). RGB is an additive process, from no color (black) to all colors (white), whereas CMYK is a subtractive process from white (reflecting all colors) to black (no light reflection).

It's easy to understand how we can produce black, white, red, green and blue with LEDs. Just turn them all ON or, either individually or collectively. If they are turned ON in pairs, we can also get cyan, yellow and magenta. Any other colors require percentages of something other than zero or 100%. Luckily, the WS2811 LED driver, discussed previously, uses PWM outputs. The PWM values are 8-bit and correspond to color chart values. This allows each LED to be adjusted to some percentage of full.

PWM values other than 0 or 255 are required for intermediate colors, such as orange or brown. Having PWM control also allows any color to be faded to black, by lowering the values of the RGB LEDs to zero while retaining the same color proportions. NeoPixels want to receive the color values for each LED in a red, green, blue sequence, so the table's 27 consecutive memory locations are defined as R, G and B for each of the LEDs 1-8.

SWITCH INPUTS

For now, I'll be using each of the eight switches to set the nine LEDs to a different color. This will test out both the switches and the NeoPixels. So, let's look at those switches. The MCU requires certain functions to be on specific pins, so it's not unusual that the leftover pins, used as switch inputs, are not on the same port. The first order of business is to gather the state of each input into a single byte, where switches 1-8 correspond to bits 0-7 of the register, *NewSwitch*. A second register, *LastSwitch*, will retain the previous sample's switch states (originally initialized to 0xFF).

A byte of 0xFF means that all switches are "not pushed." When a switch is pushed, it pulls its input to ground, and samples as a "0." By comparing these two registers (XORed), we can determine if any switch has changed state. By comparing this value with *NewSwitch* (ANDed) we can eliminate the push changes, keeping only those changes due to releasing a switch. The complement of this will be indicated with a "0" any key that has been released since the last sample. This is combined (ANDed) with *COSSwitch*, which

ABOUT THE AUTHOR

Jeff Bachiochi (pronounced BAH-key-AH-key) has been writing for *Circuit Cellar* since 1988. His background includes product design and manufacturing. You can reach him at:

jeff.bachiochi@imaginethatnow.com or at:

www.imaginethatnow.com.

```
  btfss         STATUS, C       ; skip next if carry=1
  goto     TIMER1_0
;
TIMER1_1
    bsf      LATB,   Sout
    nop                           ; 125ns
    nop                           ; 250ns
    nop                           ; 370ns
    nop                           ; 500ns
    bcf     LATB,   Sout          ; 625ns
    goto     TIMER1_Continue
;
TIMER1_0
    bsf     LATB,   Sout
    nop                           ; 125ns
    nop                           ; 250ns
    bcf     LATB,   Sout          ; 375ns
    goto    TIMER1_Continue
;
TIMER1_Continue
```

LISTING 1
TIMER1 interrupt routine handles the timing of each bit to the NeoPixels. The ON time is controlled by hard coding NOPs for each one-instruction cycle delay. An instruction cycle is 32MHz /4 = 125ns.

keeps a running tally of any changes.

While we have both I's and O's to deal with, no actual work will be done here, other than sampling the status of the switches and setting the color of each LED, based on the color table. What to do with the switch status and what colors to set each LED will be handled by a second circuit. So, we'll need to set up a communications interface.

The SPI port will be used as a slave device for communication with another circuit. The UART will serve as a debug port for messages. The UART could be connected to an LCD or a PC to display messages about the status of this device. Which switch was just released? What SPI data are being received from a master? What is the bitstream going to the NeoPixels? While none of this is necessary for the operation, my coding mistakes are easier to find when I have good feedback about what's actually happening inside this black box.

I've started this project with the I/O slave device, and have not yet produced a master device. So how can I test this part of the project? I can use the trusty Arduino as a master device, and write a simple program to collect switch information and set the color table through the Arduino's SPI port. This project will use multiple slave devices, so I'm going to make use of the slave select (SS) line to choose which slave device I want to communicate with. I could have chosen to use I^2C for inter-board communication, in which case each device would require a separate address. While SPI requires more than the two signal lines of I^2C, each slave device has its own slave select input, so the code can stay identical for each without having to assign a different address to each slave device.

TESTING 1, 2, 3, 4...

The master provides a clock for the SPI shift register in each device. SPI communication transfers a byte in both directions at the same

```
int writeSwitchStatusCommand()
{
  // take the chip select low to select the device:
  digitalWrite(chipSelectPin, LOW);
  // send the device the 6-bit address register you want to write to, receive switch status
  int result = SPI.transfer(0x00);
  // send the value you wish to write to the addressed register, receive dummy
  SPI.transfer(~result);
  // take the chip select high to de-select:
  digitalWrite(chipSelectPin, HIGH);
  // return the result:
  Serial.println("Received " + String(result,HEX));
  return (result);
}

void writeLEDColorTableCommand()
{
  // take the chip select low to select the device:
  digitalWrite(chipSelectPin, LOW);
  // send the device the 6-bit address register you want to write to, receive switch status
  SPI.transfer(0x01);
  for(int i=0; i<9; i++)
  {
    Serial.print("LED" + string(i+1));
    for(int j=0; j<3; j++)
    {
      // send the value you wish to write to the addressed register, receive dummy
      SPI.transfer(FaceArray[(i*3)+j]);
      Serial.print(",");
      Serial.print(FaceArray[(i*3)+j],HEX);

    }
    Serial.println();
  }
  // take the chip select high to de-select:
  digitalWrite(chipSelectPin, HIGH);
}
```

LISTING 2
Code listing showing the SPI transfers

time. How the data are used is entirely up to you. I found a resource from ST Microelectronics that explained a simple protocol they use for some of their products [4]. The first two MSBits (most significant bits) indicate one of four modes of operation: Read, Write, Read and Clear Status, and Read Device Information. The last 6 bits indicate a register number. With this protocol, I could individually change any of the 27 registers used in the LED1:8 color table. At this point, I think I only need to write data, but it won't hurt to follow this suggested protocol.

Because data moves in both directions at the same time, you can see that a slave can't possibly know what to send before a request is made (the slave receives a command), so there will be times when "dummy" data is sent just to fill the gap. After sending a "read" command, the master must send a dummy byte to allow the slave to respond with the data and clock it back to the master.

There is only the time between clock cycles to determine which register is requested and transfer it into the *SSP1BUF* register, before the master's clock begins shifting data out. This is not an issue when a slave device like an EEPROM has the hardware to handle it. But when the slave is a software device, there is code to execute. These data might be unobtainable in the required time frame, because the master doesn't idle between bytes. You may need to expand your protocol by 1 byte to allow for this. The master might need to send 3 bytes: request, dummy, dummy. The slave would send dummy (while waiting for the request), dummy (while it processes the request) and then the data.

In this case I can make use of the first exchange by always loading the switch status into the SPI buffer before the beginning of

```
//***************************************
// request switch status
//  if a switch bit = 0, then fill
//  the array with the appropriate color
// pause
//***************************************


void loop()
{
  switches = writeSwitchStatusCommand();
  if (!(switches & 1))
  {
    if(debug&1)
    {
      Serial.println("Switch 1");
    }
    Fill(Black);
  }
  if (!(switches & 2))
  {
    if(debug&1)
    {
      Serial.println("Switch 2");
    }
    Fill(Red);
  }
  if (!(switches & 4))
  {
    if(debug&1)
    {
      Serial.println("Switch 3");
    }
    Fill(Orange);
  }
  if (!(switches & 8))
  {
    if(debug&1)
    {                              (continues)
```

```
(Listing3 continued)
      Serial.println("Switch 4");
    }
    Fill(Yellow);
  }
  if (!(switches & 16))
  {
    if(debug&1)
    {
      Serial.println("Switch 5");
    }
    Fill(Green);
  }
  if (!(switches & 32))
  {
    if(debug&1)
    {
      Serial.println("Switch 6");
    }
    Fill(Blue);
  }
  if (!(switches & 64))
  {
    if(debug&1)
    {
      Serial.println("Switch 7");
    }
    Fill(Violet);
  }
  if (!(switches & 128))
  {
    if(debug&1)
    {
      Serial.println("Switch 8");
    }
    Fill(White);
  }
  delay(100);
}
```

LISTING 3
As shown here, each bit position 0-7 (switches 1-8) has been assigned a color: black, red, orange, yellow, green, blue, violet and white.

any communications—normally a wasted dummy byte from the slave. If we can receive switch status on every transfer, we only need to be able to write data to the slave. A command of "0" would be a write to address "0" (switch status register). One additional byte includes the data to write to the slave. This byte will be the complement of the switch status it just received, and will then be IORed with *COSSwitch* (switch status register) to reset (and acknowledge) the status. A command of "1" would be a write to address "1" (LED color table start). This would be followed by (27) bytes to fill the LED Color Table, and that table's data would

```
//**************************************
// fill array of bytes with color grb
// then send array using SPI
//**************************************
void Fill(long grb)
{
  for (int i=0; i<9; i++)
  {
    //Serial.println(grb,HEX);
    int x = grb/0x10000;
    FaceArray[i*3] = x;
    //Serial.println(x,HEX);
    int y = (grb - (x * 0x10000)) / 0x100;
    FaceArray[(i*3)+1] = y;
    //Serial.println(y,HEX);
    int z = grb - (x * 0x10000) - (y * 0x100);
    FaceArray[(i*3)+2] = z;
    //Serial.println(z,HEX);
  }
  writeLEDColorTableCommand();
}
```

LISTING 4
The Fill(long grb) function breaks the 32-bit value into 4 bytes.

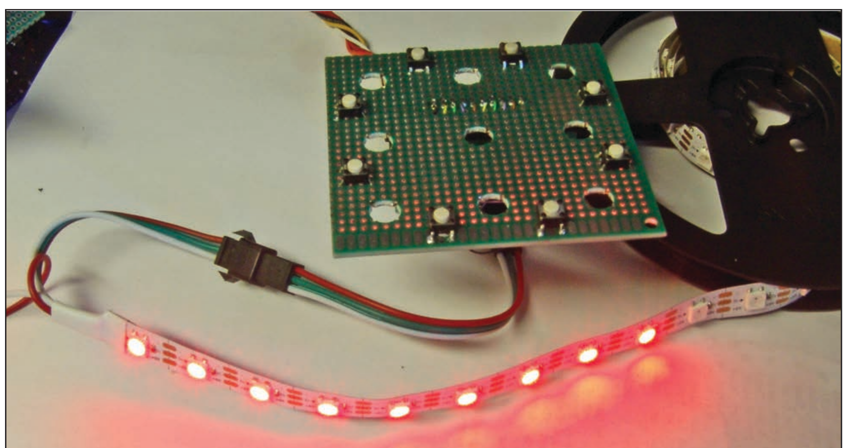


FIGURE 6
Prototype of this month's circuit in operation

Additional materials from the author are available at:
www.circuitcellar.com/article-materials
References [1] through [4] as marked in the article can be found there.

RESOURCES

Adafruit | www.adafruit.com

Microchip Technology | www.microchip.com

ST Microelectronics | www.st.com

then be used to update the NeoPixels. **Listing 2** shows the code to handle these two transfers.

Our circuit samples the switches while its idle. Any button pushed and released is added to the switch status register *COSSwitch* as a zero in the appropriate bit position (switch 1-8 = bits 0-7). The Arduino's loop function begins with a request of the switch status. It receives this by using the command 0x00 (write to address 0). Each bit position 0-7 (switches 1-8) has been assigned a color: black, red, orange, yellow, green, blue, violet and white. When a zero is found in a bit position, the fill routine is called with the bit's associated color palette. Then the function ends with a short delay before looping back (**Listing 3**).

The `Fill(long grb)` function breaks the 32-bit value into 4 bytes (**Listing 4**). The MSByte (most significant byte) isn't used, but the second byte becomes the red color value, the third the green and fourth the blue palette values for that particular color. An array that holds the 3 color bytes for each of the nine LEDs is filled with that color. There isn't any reason each LED couldn't have a different color.

This array is used by the `writeLEDColorTableCommand()` function to update our PCB via SPI, with data for its `LED Color Table`. When our board receives these 27 bytes, it stores them in the `LED Color Table` and produces NeoPixel serial data to update the nine LEDs with new color data.

It took longer to wire up a 6-pin connector to the Arduino than it did to write this simple test program. That's what I like about the Arduino—it makes a great testing vehicle. Each switch will, in turn, change all the NeoPixels to their particular colors (**Figure 6**).

THE BIGGER PICTURE?

This column introduced a circuit that will program a string of nine NeoPixels to specific colors, based on receiving SPI data from some master SPI device. In addition, the circuit has eight push buttons that are constantly scanned for user presses. Switch status is reported to the master SPI device when it is requested. In our test case, the master merely sent data to change all LEDs to a particular color, based on the switch status. Remember—this is just a small part of a larger conglomeration, which will replace an inexpensive and simple, yet perplexing, curio with a pricey technology-ridden one.

I've designed using the SK6812 or WS6812 in its IC form, however these are also available pre-mounted on flex circuit and sold by the meter (Figure 6). I don't want to give too much away here, so I'll leave you with something to think about. Put on your Sherlock Holmes "deerstalker," and see if you can figure it out before next month. Too much to learn, so little time.

When it comes to robotics, the future is now!

From home control systems to animatronic toys to unmanned rovers, it's an exciting time to be a roboticist. *Advanced Control Robotics* simplifies the theory and best practices of advanced robot technologies, making it ideal reading for beginners and experts alike. You'll gain superior knowledge of embedded design theory by way of handy code samples, essential schematics, and valuable design tips.

With this book, you'll learn about:

- **Communication Technologies**
- **Control Robotics**
- **Embedded Technology**
- **Programming Language**
- **Visual Debugging... and more**

Get it today at cc-webshop.com.

PRODUCT NEWS

2,000W Modular Power Supplies Offer Full MoPP Isolation

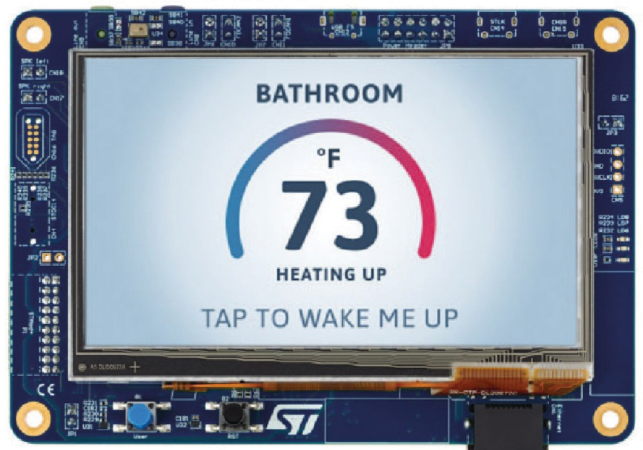
TDK has announced its TDK-Lambda brand QM8B modular power supplies rated at up to 2,000W. This further extends the QM series which can provide 550W to 2,000W output power. The QM8B models are available with up to 18 outputs, have full MoPP (Means of Patient Protection) isolation and low acoustic noise. With medical and industrial safety certifications, the power supplies are suitable for use in medical, test and measurement, communications and broadcast equipment. This avoids the need for multiple power supplies in systems requiring a large number of independent voltages.

Accepting a wide range 90 to 264Vac, 47-63Hz input (440Hz with reduced PFC), the QM8B can deliver 1200W at low line and 2000W with a high line 180-264 Vac input. With its modular construction, the series can be configured using a simple on-line configurator to provide 1 to 18 independently regulated outputs and include individual output good signal and remote on/off functions. The QM series module output voltages range from 2.8V to 105.6V and have output power levels from 300W to 1200W. Overall case dimensions for the QM8B are 200mm × 63.3mm × 268 mm (W x H x D). The QM8B will operate in ambient temperatures of -20 to +70°C (-40°C start-up), with output power and output current linearly derating above 50°C to 50% at 70°C.

TDK-Lambda | www.tdk-lambda.com

UI Software Framework for STM32 MCUs Gets Upgrade

STMicroelectronics (ST) has updated the TouchGFX user-interface software framework for STM32 microcontrollers, adding new features that enable smoother and more dynamic user interfaces and lower demand on the memory and CPU. TouchGFX is a free tool in the STM32 ecosystem. Comprising two parts—TouchGFX Designer PC tool for designing and configuring rich user interfaces, and TouchGFX Engine software that runs on the end-device to secure high UI performance—the latest version 4.12 contains updates to both. Users can now build sophisticated user interfaces on one-chip display solutions without external RAM or flash, save power for longer battery life and benefit from easier development to get to market faster.



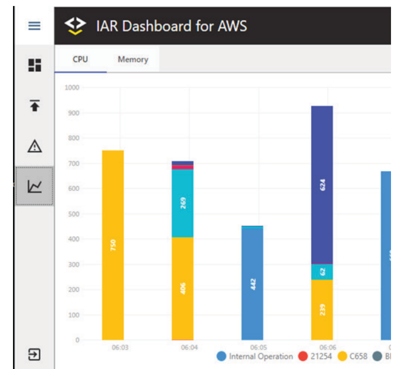
In TouchGFX Engine, a partial framebuffer mode now allows the buffer to operate using as little as 6KB of RAM. A fully functioning user interface can now have just 16KB of RAM, so that small STM32 MCUs can deliver great user experiences without external memory. The updates to TouchGFX Designer include extensions to the powerful set of customizable widgets, adding features such as Scale and Rotate that increase the power of simple drag-and-drop programming. The complete TouchGFX Suite, including TouchGFX Designer and TouchGFX Engine, is available to download free of charge from www.st/com/touchgfxdesigner.

STMicroelectronics | www.st.com

Embedded Toolchain for Arm Provides Integration with AWS

IAR Systems has launched a new edition of IAR Embedded Workbench for Arm that provides integration with Amazon Web Services (AWS). IAR Embedded Workbench for Arm, AWS edition, provides developers with the possibility to log in to an AWS account from within the C/C++ development toolchain IDE. During debugging, they are able to access the TCP/IP interface, see the status of MQTT packages and inspect the device shadow for complete control from device to cloud. The cloud communication with AWS IoT Core can also be inspected and controlled by subscribing to AWS topics and publishing commands, says IAR Systems.

The toolchain also provides support for the IoT Realtime Operating System, Amazon FreeRTOS. Based on the FreeRTOS kernel, Amazon FreeRTOS includes software libraries which make it easy to securely connect devices locally to AWS Greengrass, or directly to the cloud, and update them remotely. For new devices, developers can choose to build their embedded and IoT application on a variety of qualified microcontrollers from companies collaborating with AWS and IAR Systems, including NXP, STMicroelectronics and Texas Instruments.



IAR Systems | www.iar.com

PRODUCT NEWS

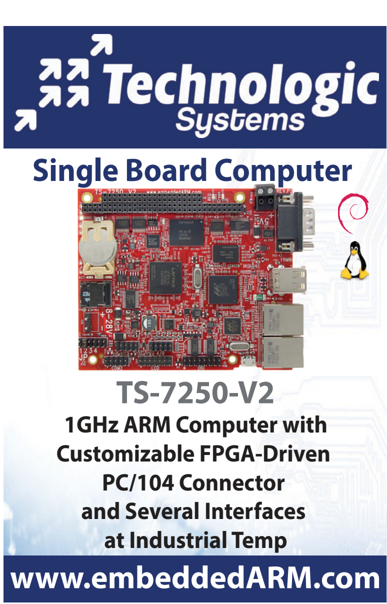
IDEA BOX

The Directory of PRODUCTS & SERVICES

AD FORMAT:
Advertisers must furnish digital files that meet our specifications (circuitcellar.com/mediakit).

All text and other elements MUST fit within a 2" x 3" format.
E-mail adcopy@circuitcellar.com with your file.

For current rates, deadlines, and more information contact Hugh Heinsohn at 757-525-3677 or Hugh@circuitcellar.com.



TEST YOUR EQ

Contributed by David Tweed

Problem 1— The following code for a Microchip (formerly Atmel) ATmega328 is intended to scan a 3×4 keypad and return a code indicating which key, if any, is pressed. However, there's a bug in it. Can you spot it?

```
#define KEYPAD A
#define KEYPAD_PORT PORT(KEYPAD)
#define KEYPAD_DDR   DDR(KEYPAD)
#define KEYPAD_PIN   PIN(KEYPAD)

uint8_t GetKeyPressed()
{
  uint8_t r, c;
  KEYPAD_PORT |= 0X0F;
  for (c=0; c<3; c++) {
    KEYPAD_DDR &= ~(0X7F);
    KEYPAD_DDR |= (0X40>>c);
    for (r=0; r<4; r++) {
      /* If keys pressed, return code 0-11.
       */
       if (!(KEYPAD_PIN & (0X08>>r))) return
(r*3+c);
    }
  }

  /* No keys pressed, return special code.
   */
  return 0XFF;
}
```

Problem 2— You are given a design task in which there's a digital signal carrying pulses at a rate of 1Hz to 100Hz. The requirement is to produce a logic signal that is high when the input pulse rate falls between 40Hz and 60Hz. The goal is to come up with the simplest circuit that can do this—preferably without using a microcontroller or anything other than commonly available SSI/MSI logic parts.

Problem 3— Can you think of any other applications for such a circuit?

Problem 4— The original Zilog Z80 microprocessor had about 8,500 transistors, and was produced with a minimum feature size of about 6 microns. If it were manufactured using modern 7nm transistors, how small would it be?

The Future of IoT as Safety Resource

Safer Living Through AI and IoT

By
Jen Bernier-Santarini,
Adesto

The world is increasingly afflicted by natural disasters. Almost every day we turn on the news to see fires, floods, hurricanes, tsunamis and other storms striking yet another major population center. By now, many of us—or our families and friends—have been personally affected. And while in most cases we can't yet prevent these occurrences, we can begin to better prepare for them and mitigate damage.

Disaster management is a very real area of research that predates much of today's technology, and is one that is eager to embrace its potential. Experts in this area proffer three key pieces of advice: take measures to mitigate potential damage, implement means for immediate victim assistance and plan for rapid recovery. While these pieces of advice were probably originally conceived at a time when the main actors in disaster management would be people, technology can and is now helping with all three.

In some ways, huge trends, such as AI, the IoT and Big Data, have the intensity of natural phenomena, but they have the potential to be forces for good. We can now use technology to spread alerts faster than ever before, ensuring people living in areas of risk can be better prepared to take evasive action should the need arise. Smart sensors can now supply the raw data needed to detect potential threats sooner, and high-speed networks can deliver sensor data to server farms where AI can crunch the numbers to find patterns that match threats.

GLOBAL IMPACT

But there is much work to be done. The global financial impact of natural disasters has been estimated at more than $300 billion a year and climbing, with some estimates much higher when taking downstream impacts into account. Unfortunately, according to the United Nations 2019 Global Assessment Report on Disaster Risk Reduction (GAR2019) [1], today's international development financing system allocates approximately 20 times the funding to emergency response, reconstruction, relief and rehabilitation activities compared to that allocated for disaster prevention and preparedness.

So how will IoT technologies help with prevention and preparedness? The IoT is pervasive, and its technology is becoming less expensive, which makes endpoints like smart sensors more cost-effective and relatively easy to deploy (**Figure 1**). In terms of early warning systems, we can expect more raw data to be generated in areas prone to natural disaster through various sensors to

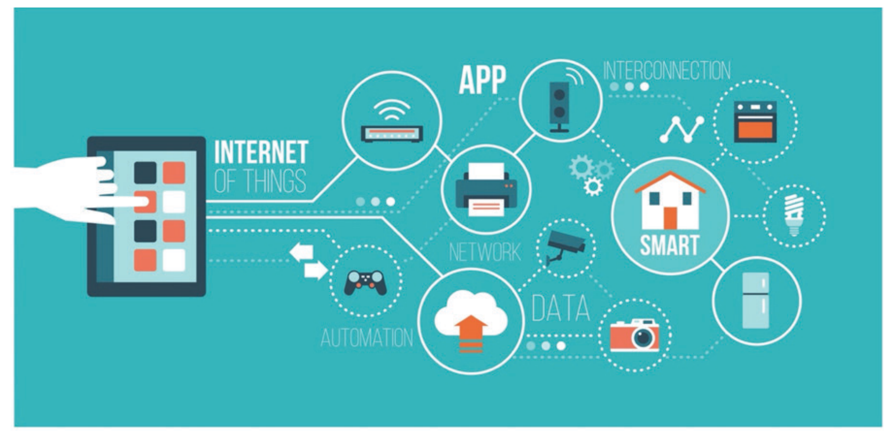


FIGURE 1
The IoT is pervasive and its technology is becoming less expensive. That's making endpoints like smart sensors more cost-effective and relatively easy to deploy.

TECH THE FUTURE

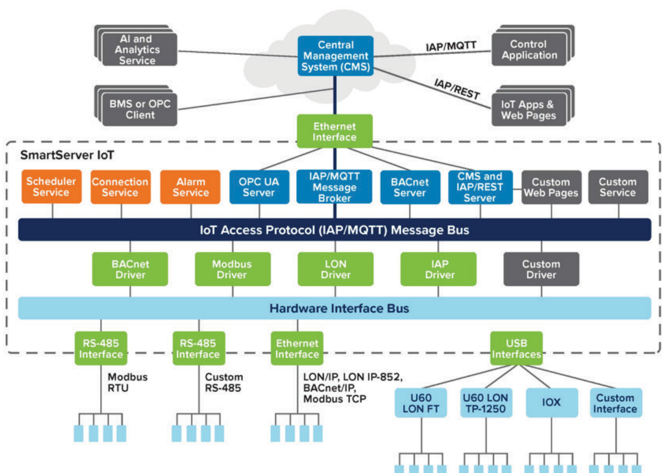


FIGURE 2
SmartServer IoT system diagram

measure earth tremors, monitor sea levels, measure carbon monoxide/dioxide levels, monitor temperature and moisture levels and more. Changes in such elements can forewarn us of imminent danger.

The data generated by each of these sensors is the key. According to the GAR2019 report, today data collection is "...often fragmented, non-universal, incommensurable and biased, and the disconnect among 'knowing' something, making it 'available and accessible' and 'applying' what is known, often remains." We see this same macro issue at work in local environments where IoT sensors are being used by municipalities and companies to gather data from various systems to create better living for citizens and employees.

Growing populations across the world are increasingly migrating to cities—many of which are rapidly turning into megacities with populations greater than 10 million people. In this environment, access to data becomes fundamental to safer living. In today's cities, IoT technology can approximate how long it will take us to drive across the city, warn us of road accidents, map our route, and help us find a parking space.

But as populations in urban areas increase, the disproportional gulf between cause and effect will become more apparent. For example, road traffic incidents may be attributable to a build-up of traffic in another part of the city, or the lack of adequate lighting on a particular street. More densely inhabited areas may generate greater potential for incidents, because the margins for error will be eroded. If one person takes a different route home it makes little or no difference to congestion; if a hundred people do it, roads can become gridlocked.

This is where the integration of disparate systems and use of AI will make all the difference. In the future, real-time data, forming seemingly incoherent patterns, will be easily analyzed by AI technologies to make traffic flow better, or reduce the potential of hazards for pedestrians and cyclists. And it will happen behind the scenes, without us having to make a conscious effort to change our natural behavior. Right now, the systems to make this work aren't seamless. These systems, even those that are connected to the internet, often exist in isolated silos.

SMART BUILDING EXAMPLE

Take a smart building as an example. Within a building, the access control systems, HVAC systems, lighting systems, elevators and other systems may all be "smart" in that they automatically turn on when needed, turn off when they aren't needed, can be monitored and adjusted remotely, but they are generally disconnected from each other. It won't be much use during an emergency if a building's emergency lights turn on, but the doors remain locked.

The reason for the disconnect is largely a legacy issue: there are so many different, un-interoperable protocols, devices and services used in existing building management systems and other industrial control systems, that integration has become a real issue. Where these systems are able to connect and work together today, it often takes vast sums of money to fund the integration effort. What we need are simple, cost-effective ways to bridge legacy systems to new IoT systems to let us make use of the valuable data that the systems generate.

The SmartServer IoT from Adesto is designed to address this issue. It makes it easier to access the wealth of data an industrial control system may hold, to enable new solutions that could make a real difference to peoples' lives. With SmartServer IoT, companies can easily connect their disparate, non-interoperable systems, devices, and services together and also connect to cloud platforms to make use of AI and predictive analytics—which can be used to understand trends and mitigate risks (**Figure 2**).

Natural disasters are potentially predictable, and manufactured incidents are often avoidable. Both rely on being able to observe, analyze and react to the world around us. And while global natural-disaster risk mitigation will require mega political, socioeconomic and cultural discussions and change, the AI and IoT technologies that can enable this change are increasingly available.

Today's technology means that we are now more equipped—through data—to defend ourselves, our homes and our possessions from harm. In the future, by bringing disparate systems together and making existing solutions more extensible, we can build even smarter and safer communities.

Jen Bernier-Santarini is VP of Corporate Communications at Adesto, a provider of application-specific semiconductors and systems for IoT. Before joining Adesto in 2019, Jen led technology communications for IP provider Imagination Technologies. With more than 25 years working in semiconductors and related technologies, her expertise includes electronic design automation (EDA) tools, connectivity technologies, processors and IP, flash memory and other off-the-shelf chips.

For detailed article references and additional resources go to: www.circuitcellar.com/article-materials
Reference [1] as marked in the article can be found there.

RESOURCE
Adesto Technologies | www.adestotech.com

PROTOTYPES
IN 4 DAYS FROM TAIWAN

No Tariffs!
Best Quality and Price!

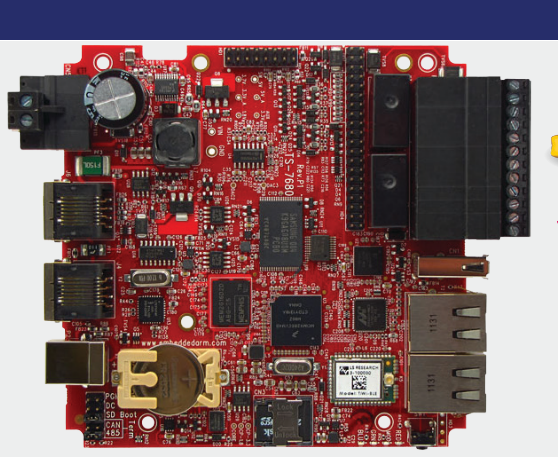
Technology:
Up to 50 Layers
Any Layer HDI
Sequential Lamination
Blind / Buried Vias
Laser Drilling / Routing
Heavy Copper

Materials:
Fr4
Metal Core
Isola
Rogers
Polyimide - Flex
Magtron

www.PCB4u.com sales@PCB4u.com
SAM & ITAR Registered UL E333047 ISO 9001 - 2015

www.embeddedARM.com

FROM THE DEEP BLUE SEA TO THE WILD BLUE YONDER

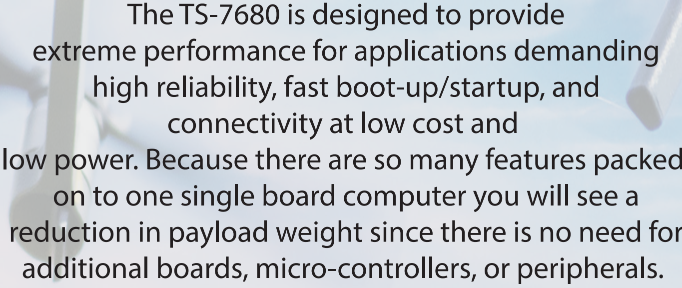


TS-7680

Low Power Industrial Single Board Computer with WiFi and Bluetooth

$159

Qty 100



The TS-7680 is designed to provide extreme performance for applications demanding high reliability, fast boot-up/startup, and connectivity at low cost and low power. Because there are so many features packed on to one single board computer you will see a reduction in payload weight since there is no need for additional boards, micro-controllers, or peripherals.

Rated for industrial temperature range of -40°C to +85°C the TS-7680 is deployed in fleet management, pipeline monitoring, and industrial controls and is working in some of the most demanding places on Earth.

The TS-7680 will help you perform at your very best in a variety of critical missions.