



circuitcellar.com

circuit cellar

Inspiring the Evolution of Embedded Design

TECH FEEDS DESIGN NEEDS FOR DIGITAL SIGNAGE



Product Focus: PC/104 Boards Trends in Code Analysis Tools |

PIC32 Does Real-Time Stock Monitoring | Capacitive vs. Inductive Sensing |

Robotic Arm Plays Beer Pong | Transistor Basics for Today's Engineer

Fun with the Itty Bitty Kit | Pressure Sensors | Filtering with Teensy 3.6 |

Attacking USB Gear with EMFI The Future of Safe Programming



Together, we are making security implementation easier for you.



Industry-leading development tools and ground-breaking security technology

IAR Systems and Secure Thingz are teaming up to make security implementation part of the development workflow with the release of Embedded Trust and C-Trust.

Embedded Trust is a security development environment, which simplifies the configuration of security, from the root of trust and key storage for a connected device to the creation of security profiles and projects.

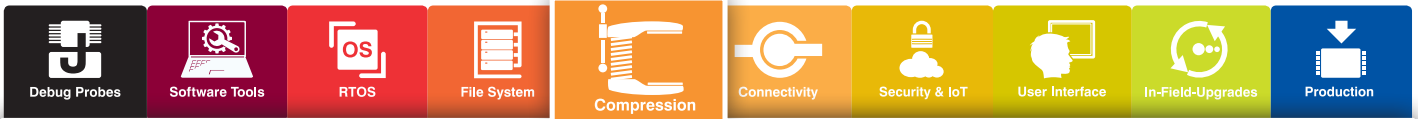
C-Trust is an extension to IAR Embedded Workbench that enables application developers to deliver secure, encrypted code as part of their standard workflow.

Make security a natural part of your day-to-day development

Learn more at iar.com/security

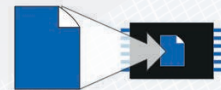


The Embedded Experts



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



VOICE COIL



LIS LOUDSPEAKER
INDUSTRY
SOURCEBOOK

SUPPORTING COMPANIES

Accutrace, Inc.	C3
AdaCore	29
All Electronics Corp.	77
CCS, Inc.	77
congatec, Inc.	67
Elma Electronic, Inc.	27
EzPCB	51
HuMANDATA, Ltd.	19
IAR Systems, Inc.	C2
Mentor, A Siemens Business	51
Measurement Computing Corp.	59
Noritake Co., Inc.	15
RoboBusiness 2019	23
SEGGER Microcontroller Systems	1
Slingshot Assembly	13
Sensors Expo & Conference 2019	55
Technologic Systems, Inc.	C4, 77

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

EDITOR-IN-CHIEF
Jeff Child

ADVERTISING COORDINATOR
Nathaniel Black

CONTROLLER
Chuck Fellows

SENIOR ASSOCIATE EDITOR
Shannon Becker

ADVERTISING SALES REP.
Hugh Heinsohn

FOUNDER
Steve Ciarcia

TECHNICAL COPY EDITOR
Carol Bower

PROJECT EDITORS
Chris Coulston
Ken Davidson
David Tweed

GRAPHICS
Grace Chen
Heather Rennae

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 346 May 2019 | 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

INPUT Voltage

It's clear that Bluetooth technology has become part of our everyday modern lives. When I say "our" I mean mostly other people. As an old curmudgeon, I'm still quite attached (literally) to the wired headphones that I use daily. I'll eventually make the shift, I'm sure. My daughter has a winter hat with Bluetooth headphones built in and that seems pretty cool. Bluetooth was slow to reach me in the automotive side of life as well—and that's because I'm not a car person.

I've noticed throughout my life is that there's a distinct difference between someone who is a "car person" or "not a car person." I am most definitely am a "not." Car people seem to get excited about choosing and buying a new vehicle and have a keen interest in the different models and brands—and can easily identify them. But that's never been me. That's probably why our two family cars are a 1997 model and a 2005 model. Because we have teenage drivers now, last year we added a third car—a used 2010 model. Of course, none of the three have embedded Bluetooth technology.

While I'm indifferent about cars themselves, I do care about being able to play music from my phone in the car. Until a few years back, I was among the last people on the planet that had used one of those audio jacks that plays from a car's tape cassette deck. Now all three of our cars have one of those aftermarket Bluetooth transmitters that plug into the car's cigarette lighter socket. (Can I still call them that?) Every time the car is started, the transmitter, through the car's speakers, says: "Bluetooth connected. The voltage is normal" and in an embarrassingly loud voice. That's my daily reminder that Bluetooth is alive and among us.


Covering Internet of Things (IoT) technologies in recent years, Bluetooth—and Bluetooth Low Energy (BLE) in particular—is most definitely front and center of any discussion of chips and connectivity. Bluetooth is on track to be as much of a machine-to-machine wireless interconnect technology as it's been an end user, consumer wireless solution.

For its part, the Bluetooth Special Interest Group (SIG) continues to expand the capabilities for Bluetooth with new features and subsets. In January, for example, the SIG announced a direction finding feature that allows devices to determine the direction of a Bluetooth signal. This capability is expected to enable the development of Bluetooth proximity solutions that can understand

device direction as well as Bluetooth positioning systems that can achieve down to centimeter-level location accuracy.

According to the Bluetooth SIG, Bluetooth location services solutions generally fall into two categories: proximity solutions and positioning systems. Proximity solutions currently use Bluetooth to understand when two devices are near each other, and approximately how far apart. They include item finding solutions such as personal property tags, as well as point-of-interest (PoI) information solutions like proximity marketing beacons. By including the new direction-finding feature, Bluetooth proximity solutions can add device direction capability. As an example, an item finding solution could not only let a user know when a personal property tag is nearby, but also in what direction.

Meanwhile, positioning systems use Bluetooth to determine the physical location of devices and include real-time locating systems (RTLS). This includes those used for asset tracking, as well as indoor positioning systems (IPS)—like those for indoor wayfinding. Today's Bluetooth positioning systems can achieve meter-level accuracy when determining the physical location of a device. But when you add the new the direction finding feature, these positioning systems could improve their location accuracy down to the centimeter-level. The direction finding feature is included in version 5.1 of the Bluetooth Core Specification, which is available to developers on the Bluetooth SIG website. In addition, Launch Studio, the Bluetooth SIG tool used to qualify new products, has been updated to support this feature.

In so many ways, Bluetooth technology is an on-going success story that's still being written. I look forward to the day when Bluetooth direction finding will point me toward where I left my headphones. 



Jeff Child

COLUMNS

40 **PC/104 Boards** Legacy That Stacks Up

PRODUCT FOCUS

By Jeff Child

44 **Embedded System Essentials** **Attacking USB Gear with EMFI** Pitching a Glitch

Embedded System Essentials

Attacking USB Gear with EMFI

Pitching a Glitch

By Colin O'Flynn

52 **The Consummate Engineer** **Pressure Sensors**

The Consummate Engineer

Pressure Sensors

Terminologies and Technologies

By George Novacek

56 **Picking Up Mixed Signals** **Fancy Filtering with the** **Teensy 3.6**

Picking Up Mixed Signals

Fancy Filtering with the

Teensy 3.6

Arm-ed for DSP

By Brian Millier

68 **From the Bench** **An Itty Bitty Education**

From the Bench

An Itty Bitty Education

STEM at Home

By Jeff Bachiochi

79 **The Future of Safe Programming** **How Programming Languages** **Evolve to Reduce Risks**

TECH THE FUTURE

The Future of Safe Programming

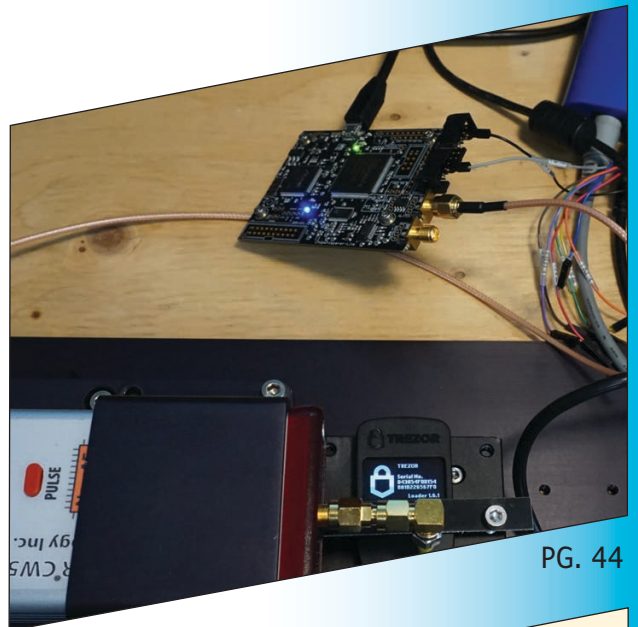
How Programming Languages

Evolve to Reduce Risks

By Quentin Ochem

76 : PRODUCT NEWS

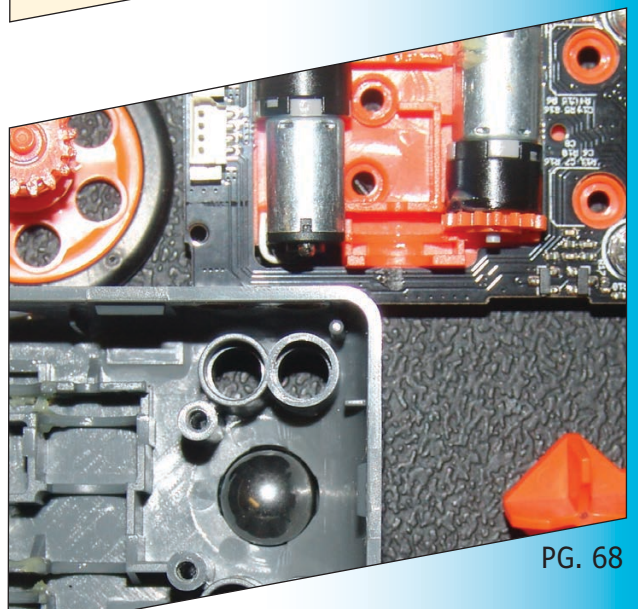
78 : TEST YOUR EQ



PG. 44

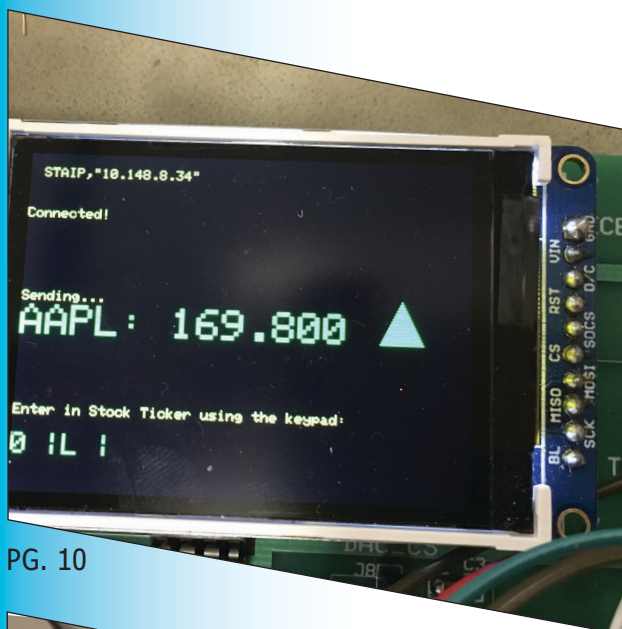


PG. 56

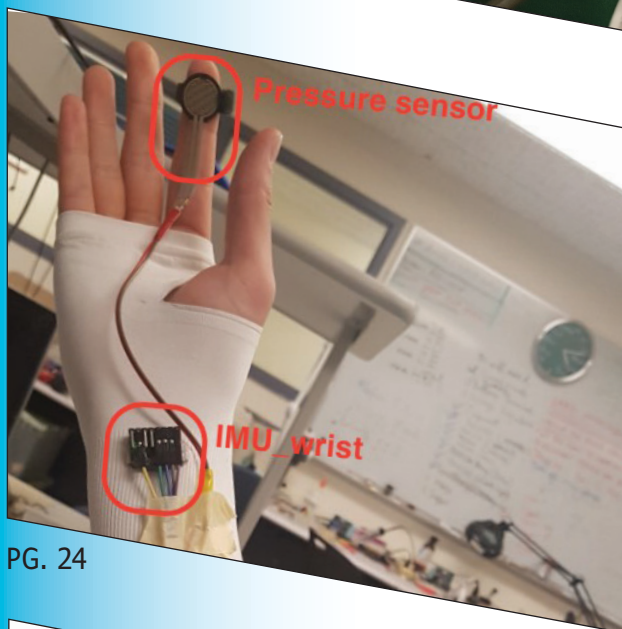


PG. 68

FEATURES



PG. 10



PG. 24



PG. 30

6 Capacitive vs. Inductive Sensing

Touch Trade-Offs

By Nishant Mittal

10 PIC32 Tames Real-Time Stock Monitoring

Market Matador

By David Valley and Saelig Khattar

16 Transistor Basics

And Their Role Today

By Stuart Ball

24 Robotic Arm Plays Beer Pong

Using PIC32s and IMUs

By Daniel Fayad, Justin Choi and Harrison Hyundong Chang

30 Digital Signage Technologies Gain Momentum

SPECIAL FEATURE

System Solutions

By Jeff Child

36 Code Analysis Tools Up Their Game

TECHNOLOGY SPOTLIGHT

Quest for Code Quality

By Jeff Child

Capacitive vs. Inductive Sensing

Touch Trade-Offs

By
Nishant Mittal

Touch sensing has become an indispensable technology across a wide range of embedded systems. In this article, Nishant discusses capacitive sensing and inductive sensing, each in the context of their use in embedded applications. He then explores the trade-offs between the two technologies, and why inductive sensing is preferred over capacitive sensing in some use cases.

Touch sensing was first implemented using resistive sensing technology. But resistive sensing had a number of disadvantages, including low sensitivity, false triggering and shorter operating life. All of that discouraged its use and led to its eventual downfall in the market.

Today whenever people talk about touch sensing, they're usually referring to capacitive sensing. Capacitive sensing has proven to be robust not only in a normal environmental use cases but also underwater, thanks to its water-resistant capabilities. As with any technology, capacitive sensing comes with a new set of disadvantages. These disadvantages tend to more application-specific. That situation

opened the door for the advent of inductive sensing technology.

In this article, we'll discuss capacitive sensing for embedded applications and how it can be used in various applications. We will then explore the use of inductive sensing in embedded products and why inductive sensing is preferred over capacitive sensing in some use cases. Finally, we'll compare the advantages of inductive sensing over capacitive sensing in these applications.

CAPACITIVE SENSING FOR EMBEDDED

Capacitive sensing operates on the principle of monitoring the change in parasitic capacitance due to a finger touch (**Figure 1**). Capacitive sensing has been used primarily in two different forms: self-capacitance and mutual-capacitance. In self-capacitance mode, the net capacitance due to a finger touch and board capacitance is additive. This capacitance includes PCB traces and PCB materials like FR4, which has more capacitance compared to Flex materials and many similar dielectrics. Self-capacitance mode is useful in general touch application like buttons for touch-and-respond applications. In contrast, mutual capacitance is well-suited for applications involving more complex sensing such as gestures, multi-touch and sliders.

Mutual capacitance sensing uses two different lines: TX(Transmitter) and RX(Receiver). The Transmitter sends a PWM signal with respect to the system V_{DD} and GND. The Receiver detects the amount of charge received on the RX electrode.

One of the difficult use cases of capacitive sensing is that it cannot operate perfectly

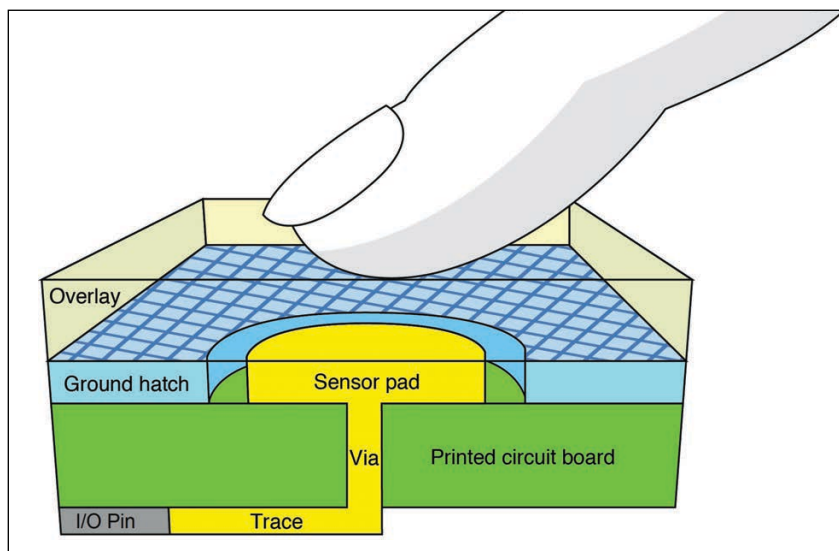


FIGURE 1
Capacitive sensing technique

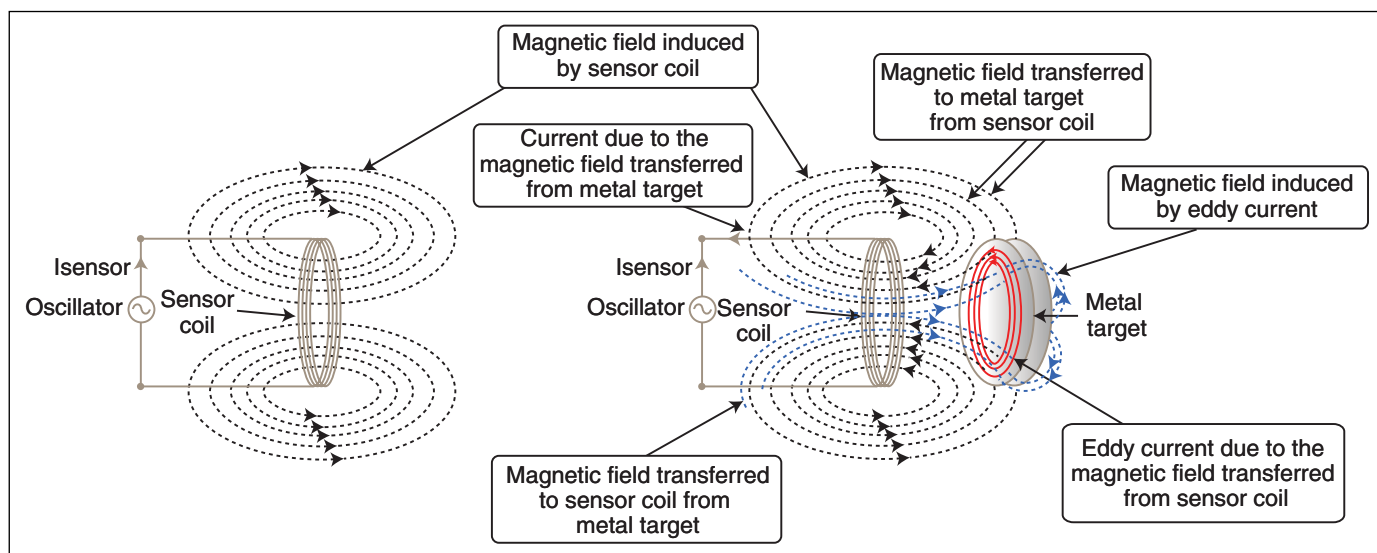


FIGURE 2
Inductive sensing technique [1]

underwater. It also requires relatively strict design guidelines to be followed for error-free operation. Capacitive sensing performance is also impacted by nearby LEDs and power lines on PCBs. Implementing auto-tuning with variation in trace capacitance, variation in capacitive sensing buttons and different slider sizes and shapes all require different designs. Implementation challenges in industrial applications include using capacitive sensing with thicker glass material (display glass) and meeting capacitive sensor sensitivity requirements with those types of materials.

INDUCTIVE SENSING FOR EMBEDDED

Inductive sensing enables the next-generation of touch technology in applications involving metal-over-touch use cases such as in automotive, industrial and many embedded and IoT applications. Inductive sensing is based on the principle of electromagnetic coupling, between a coil and the target (Figure 2). When a metal target comes closer to the coil, its magnetic field is obstructed and it passes through the metal target before coupling to its origin. This phenomenon causes some energy to get transferred to the metal target—referred to as eddy current—that causes a circular magnetic field. Eddy current induces a reverse magnetic field, in turn leading to a reduction in inductance.

To cause the resonant frequency to occur, a capacitor is added in parallel to the coil to cause the LC tank circuit. As the inductance starts reducing, the frequency shifts upward changing the amplitude throughout. In contrast to a capacitive sensor, inductive sensing is able to operate reliably in the presence of water thanks to the removal of a dielectric from the sensor. This advantage brings inductive sensing touch sensing to a wide range of applications

that involve liquids such as underwater equipment, flow meters, RPM detection, medical instruments and many others. Inductive sensing also supports biomedical applications. In general applications, inductive sensing enables replacement of mechanical switches and proximity sensing of metal objects. For example, in automotive applications, inductive sensing can be used to replace mechanical handles as well as detect car proximity. Some of these examples will be discussed in detail later.

Currently, the primary design challenge for implementing inductive sensing is designing coils with 100% production yield where inductive trace spacing is very narrow,

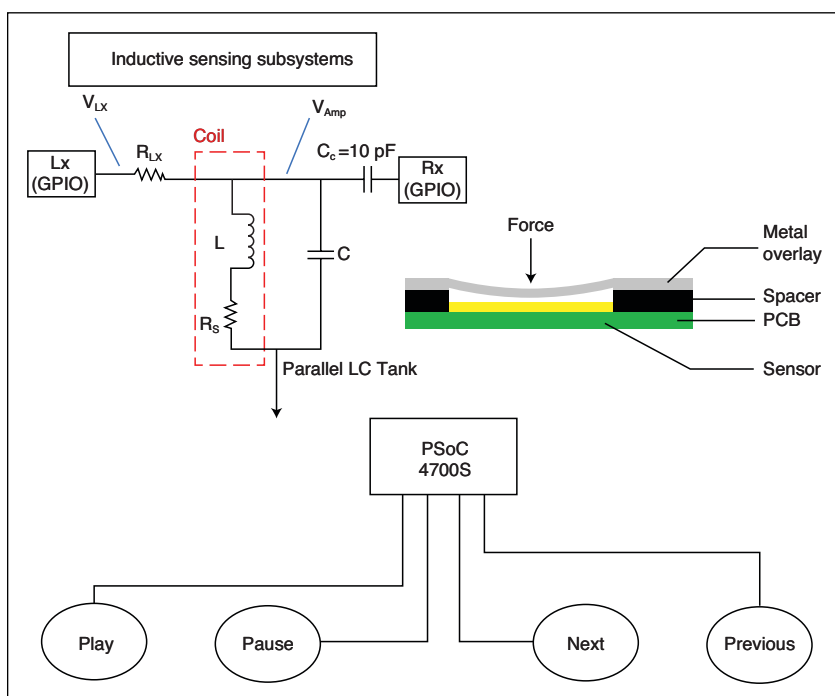


FIGURE 3
Shown here is the architecture of a water-resistant Bluetooth speaker using inductive sensing.

such as using 4-mil spacing. There is also the consideration of meeting inductive values with variations in PCB laminate materials.

USE CASES FOR EACH METHOD

Capacitive sensing is undeniably useful in a great many applications. However, for certain use cases inductive sensing offers greater reliability, ruggedness and usability.

Consider the use case of a Bluetooth speaker that needs to be water resistant and is intended for use in up to 2' underwater for half an hour. This use case requires more than just that

the product is functional underwater. It also requires that the user can adjust the speaker in these circumstances. Such operation needs to be simple, consistent and reliable—even in the presence of water.

With capacitive sensing, such operation is partially possible using mutual capacitive sensing employing complex shielding techniques. However, the device would offer a less than ideal user experience. For example, there would be inconsistent responsiveness from the touch interface. Due to changes in the dielectric introduced by the presence of water, its responsiveness would not be consistent with how the device operates when it is used out of water

For this application, metal-over-touch using inductive sensing would provide a consistent and reliable user performance (Figure 3). Alternatively, a mechanical button and/or dial could be used. However, a mechanical interface is costly compared to a coil printed on a PCB and connected to a few passive components. Additionally, a mechanical button can break or fail, providing a much shorter useable lifespan than an inductive button would.

Consider another use case employing proximity sensing: A vehicle detection system needs to monitor when another vehicle approaches within two meters and signal the driver on the dashboard or navigation panel. This functionality can be implemented using inductive sensing. A hardware board containing multiple coils at different locations using a flex cable—all around the dashboard and center of the headlight areas (Figure 4). Data from the inductive coils is collected by an inductive sensing controller such as the PSoC 4700S from Cypress Semiconductor. The controller would then analyze the data to determine the presence or absence of other cars in a 4 m vicinity around the vehicle.

Capacitive sensing could also be used for vehicle proximity sensing. Inductive sensing is rugged, environment-independent, and easy to design and develop from an engineering point of view. In addition, little tuning is required to achieve the desired closed loop for a particular application.

Note: The controller need not be placed far away from the coils to improve signal-to-noise ratio (SNR). Individual controllers can be used to optimize the design. The block diagram mentioned in Figure 4 is a principle representation.

COMPARATIVE APPROACH

In general, designing an inductive sensor is fairly straightforward (Figure 3). A typical inductive sensor requires one or more inductive coils, as determined by the requirements of the

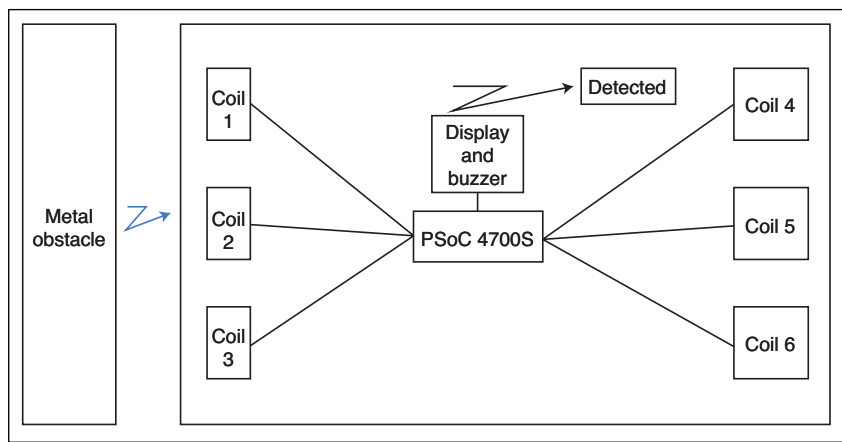


FIGURE 4

Using inductive sensing to determine vehicle proximity in an automotive application.

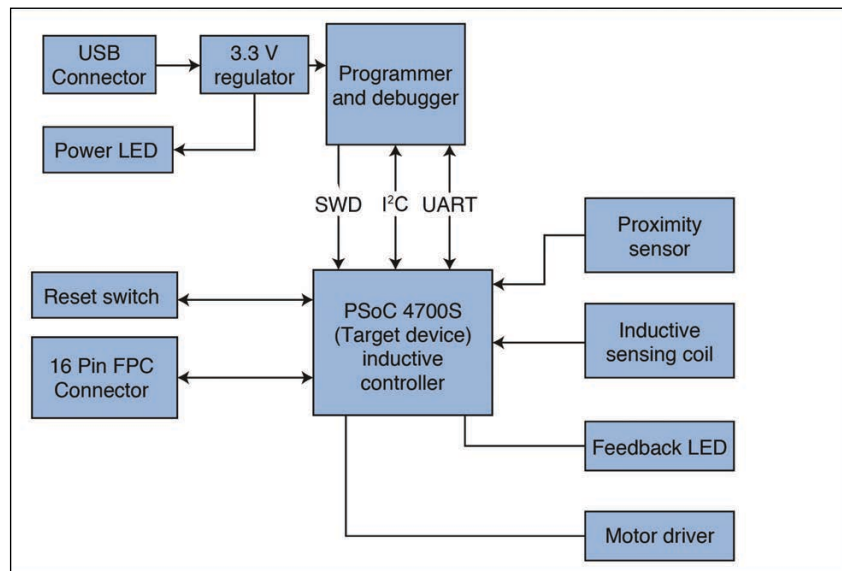


FIGURE 5

Inductive sensor block diagram

For detailed article references and additional resources go to: www.circuitcellar.com/article-materials
References [1] through [4] as marked in the article can be found there

RESOURCES

Cypress Semiconductor | www.cypress.com

application. The sensor needs to be interfaced to the controller using suitable drivers or controllers to be understood by the microcontroller. This interface can be implemented using external components. However, to reduce system design and manufacturing complexity, some inductive controllers integrate driver and converter circuitry to convert inductive sensor data into raw counts which can then be processed using suitable algorithms. To learn more about the techniques involved in designing the circuitry around inductive sensing and controller check out Cypress' Inductive Sensing Evaluation Kit product page [2].

To program the inductive sensing controller, we need a suitable programmer—either on board or using external programmers. You need to decide the maximum power to be provided. Here we have shown the system at 3.3 V, however one can range from 1.8 V to 5 V. Next, all the interfaces in the design—like LEDs, motor drivers and so forth—need to be decided and placed accordingly. **Figure 5** shows the system level block diagram of an inductive sensing board.

Figure 6 shows the design flow involved in a typical inductive sensing application. First, assess how sensitive the system needs to be. Sensitivity determines the coil size and its number of turns. The application also impacts the shape of the coil. For example, a slider interface requires a series of squares or an elongated rectangle. The next step is to calculate the tank capacitor and the inductance based on the number of turns, spacing, width and diameter. To understand the detailed steps, refer to Cypress' Inductive Sensing Design guide [3].

Capacitive Sensing on the other hand requires measurement of theoretical capacitance with the required dielectric constant. During the layout, the designer is required to follow strict layout guidelines like ground shielding—CapSense traces have to have equal length for constant C_p and so forth. For more details on CapSense design, refer to Cypress' CapSense Design Guide [4].

Once these parameters are decided, the next step is to begin the mechanical design, specifically the overlay—also known as the metal target. An overlay comprises two materials whose specifications need to be decided: the metal target and the adhesive. The metal target material determines the amount of deflection and response. I recommend using an aluminum overlay for inductive sensing application because of its better deflection and response. For button applications, a higher Newton force on the overlay causes deflection throughout the overlay, leading to undesirable false triggering throughout the coils. For this use case, the user should only be able to press the buttons just

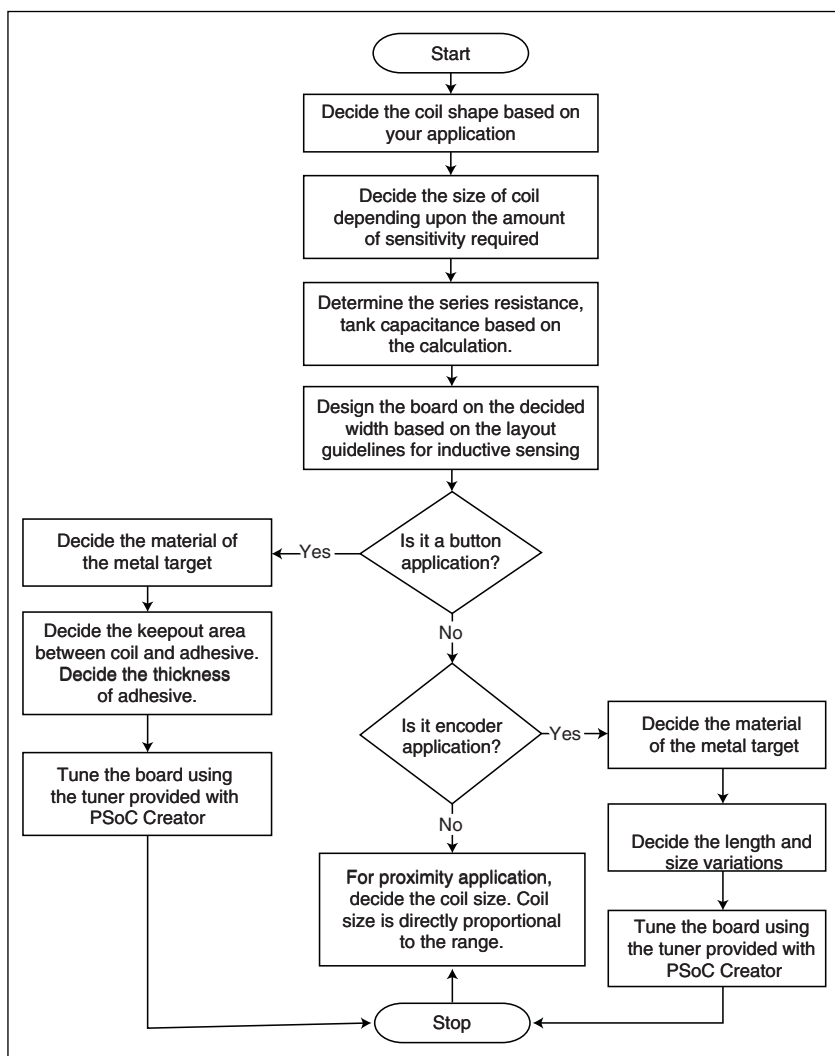



FIGURE 6

Design flow chart for a typical inductive sensing application

enough to generate feedback. Pressing the overlay harder can even deform the overlay. Once all these things are intact, the board is designed and fabricated. The advantage of PSoC Creator IDE is it provides a user-friendly Inductive Sensing Tuner GUI which can be used by designers to serve their design needs.

Both capacitive and inductive sensing enable OEMs to build intuitive, touch-based user interfaces to make their products more intuitive and easier to use. Because of its versatility, capacitive sensing has become the technology of choice in a great many applications. However, for applications where water tolerance is required, inductive sensing provides a robust and cost-effective alternative. 

ABOUT THE AUTHOR

Nishant Mittal is a Systems Engineer in Hyderabad, India.

PIC32 Tames Real-Time Stock Monitoring

Market Matador

With today's technology, even simple microcontroller-based devices can fetch and display data from the Internet. Learn how these two Cornell students built a system that can track stock prices in real time and display them conveniently on an LCD screen. For the design, they used an Espressif Systems ESP8266 Wi-Fi module controlled by a Microchip PIC32 MCU.

By *David Valley and Saelig Khattar*

We challenged ourselves to build a system using the PIC32 microcontroller (MCU) from Microchip Technology that could track stock prices in real time and display them. The goal was to create a PIC32 system that connects to the Internet and can work as a server/client to perform several functions and eventually serve as a central home hub. The system can be easily modified to fetch and display any kind of data from the Internet, as long as there is an API for it.

The rationale behind this project was that there are few libraries or applications of the PIC32 using the Espressif Systems ESP8266. Both the chips individually are highly capable, inexpensive and can be used for even large-scale manufacturing. We wanted to create a prototype PIC32 system that has Internet connectivity and can be easily extended to perform a multitude of things. We used "Protothreads", a lightweight threading library created by Adam Dunkels [1], to make our system efficient and capable of handling a variety of tasks simultaneously.

SYSTEM DESIGN

The system works as a TCP server that connects to a Python Client and fetches real-

time stock information for any company the user inputs. The input is a 12-digit keypad that works like a cell phone keyboard. The user inputs the stock symbol for a company, and the system displays the stock price on an LCD monitor, along with the corresponding arrows for increase or decrease in price relative to the last fetch. A high-level block diagram for our system design is given in **Figure 1**. In essence, we first wait for the Python Client to connect to the server. Until this happens, the server remains idle. The user can then input a stock symbol using the keyboard at any time, which triggers an API call. The display is then updated with the price of the stock, and refreshes automatically every 5 seconds.

We made use of Sean Carroll's Development Board, which contains a TFT LCD, the PIC32 MCU, several peripheral pins and support for a port expander. We connected the ESP8266 transmit and receive pins to RA1 and RB10, respectively, on the PIC, as these pins support UART. We used UART channel 2 for communication with the ESP8266. The LCD communicates with the PIC via SPI channel 1 on pin SCK1.

To use the 12-digit keyboard, we made use of a port expander. This allowed us to wire the keyboard to a series of seven consecutive

pins on the PIC, which helped us to write more concise code. Additionally, by using the port expander, we were able to refer to Bruce Land's sample code [2], which helped serve as a template for writing a bitmask lookup table. The keyboard worked by reading bits on pins RY0-RY6, which were the seven pins used by the port expander. They were then compared against the entries in the bitmask lookup table. This is how readings were made by our keyboard thread, which we discuss further in a later section of this article. The hardware schematic is shown in **Figure 2**. A link to a more detailed diagram of Sean Carroll's Development Board is available on the *Circuit Cellar* article materials webpage. We did not use external resistors for pins RY4-RY6, because the port expander has internal pull up resistors that we enabled in software. The bit readings for these three pins were then active low, so we inverted our logic to correctly register key presses.

SYSTEM COMMUNICATION

You can program and communicate with the ESP8266 Wi-Fi Module using AT Commands. Numerous AT Commands for this module are provided in the ESP8266 datasheet. See the *Circuit Cellar* article materials webpage for the link. We sent AT commands from the PIC32 to the ESP module using UART Serial communication. We would send strings (or rather, character arrays) from the PIC32 to the ESP module containing the AT Command, and would wait a response on the receive line of the PIC.

The ESP communicated with our Python Client (independent of OS) via a socket connection. The client connected to the module at its IP address on port 333 (the default when the ESP module is set up as a server). The ESP module sent this client our

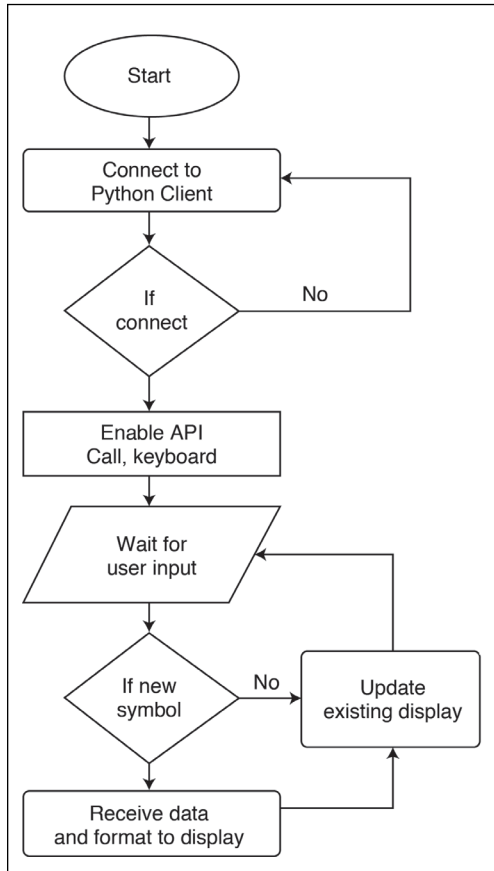


FIGURE 1

Shown here is a high-level block diagram of our system for tracking stock prices in real time.

custom commands based on user input on the keypad. The client received these commands, retrieved the necessary information (described in the next section), and sent this information back over the socket connection. The socket communication client side was handled by the standard Python socket library.

Next, our Python Client communicated with the Intrinio Web API using HTTP GET Requests. Based on the command it received from the ESP, the client made the necessary

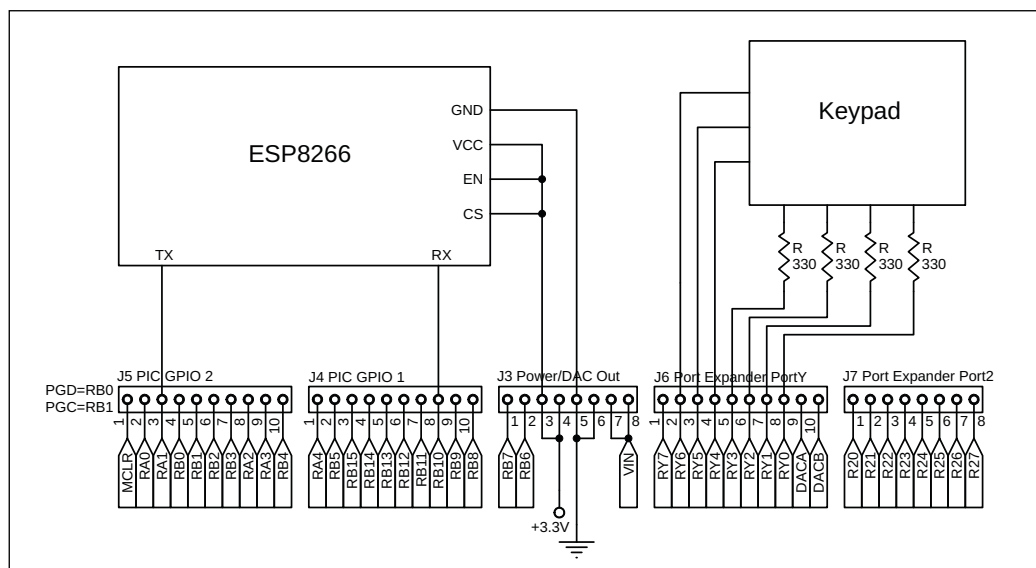


FIGURE 2

Hardware schematic. The row of connections at the bottom are the interfaces to Sean Carroll's Large Development Board PIC32MX25F128B.

GET request, formatted the response, and sent it back to the ESP module. The GET requests were formatted based on the API requirements, and contained our Intrinio API key. On an API call, Intrinio returned a JSON string with the necessary information. The client then parsed this JSON string to get the stock price. The overall communication of our project is shown in **Figure 3**.

SOFTWARE THREAD STRUCTURE

Our software for this project consisted of three distinct threads: a main thread, one to set up the ESP module and one to handle keyboard input. We also had a function to make an API call from the PIC MCU.

Main Thread: To make the software design more structured and the software workflow clearer, we developed a “main” protothread. This thread first spawned the ESP Init thread, and then waited for a connection from our client, blocking the rest of the system until this requirement was satisfied. Once it had detected the client had connected, it would then check to see if a stock symbol had been entered. If so, it made an API call which updated the corresponding stock price on the LCD in real time.

ESP Init Thread: To set up the ESP module, we had to send it various AT commands before we began to use it in our application. To check if the module worked, simply sending it “AT” via serial—which can easily be done using Putty or the Arduino Serial Monitor—and receiving the string “OK” verified its proper operation. Next, we reset the module, using

“AT+RST” to ensure any previous settings would not interfere with our current setup.

Once we ensured the module worked and was reset, the next step was to connect the module to a Wi-Fi network. We used RedRover, a free Wi-Fi network available at Cornell University. To connect this module to RedRover, we first registered the device with Cornell IT, and then sent the AT command “AT+CWJAP_DEF=RedRover” to the module. It should be noted that RedRover does not require a password to join the network, but the AT Command can accept a password argument. After it was connected, we got the IP address of the module using “AT+CIFSR.” Next, we enabled the ESP module to have multiple connections using “AT+CIPMUX=1” and configured the ESP module as a server using “AT+CIPSERVER=1”. This thread sent this series of AT commands, and was spawned once at system start.

Keyboard Thread: Our keyboard thread works by reading bits from pins RY0-RY3 on the port expander (which map to horizontal rows of the keyboard) on the PIC, and then the bits from RY4-RY6 (which map to vertical columns). The bit readings from these two groups are then ORed together and the value compared against a lookup table, which we defined locally in our thread. We would have used a different set of pins RA0-RA3 and RB7-RB9 on the PIC32. However, due to a usage conflict with other pins on the ESP module, we used a port expander.

We could not use a series of seven consecutive pins on the MCU outright, due to these conflicts with the ESP. Merely swapping single pins where conflicts existed would mean different bit masks and subsequently produce different values when the row and column values were ORed together. These would not match any of the lookup table entries. One solution to this problem was to manually calculate the expected values from these OR operations and create a new bitmask lookup table. However, we felt a cleaner solution involved using the port expander. For this, we referred to the example code in [2], which initialized and set up the port expander and the main keypad scanning logic.

To translate keypad number presses to letters, we made use of a large case statement. We had 27 cases—one for each letter of the alphabet, plus an additional space character. In our logic, once the pound sign was pressed, which worked as our “enter” button, the current value in a running buffer was passed as the argument to this case statement. The letter corresponding to the numbers in the buffer was then stored in a second ticker symbol buffer. Once a total of four characters (which is the maximum length

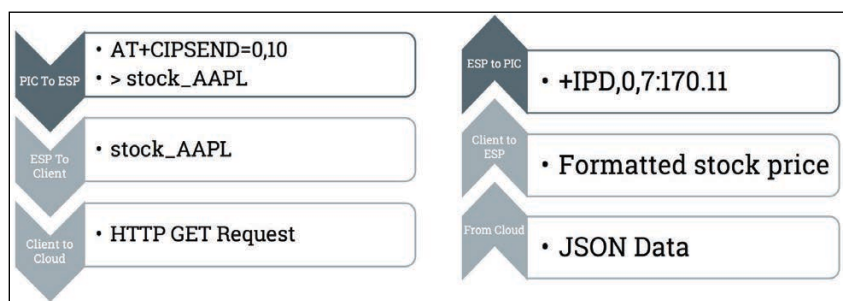


FIGURE 3

Communication from PIC to cloud

For detailed article references and additional resources go to:

www.circuitcellar.com/article-materials

References [1] through [3] as marked in the article can be found there

RESOURCES

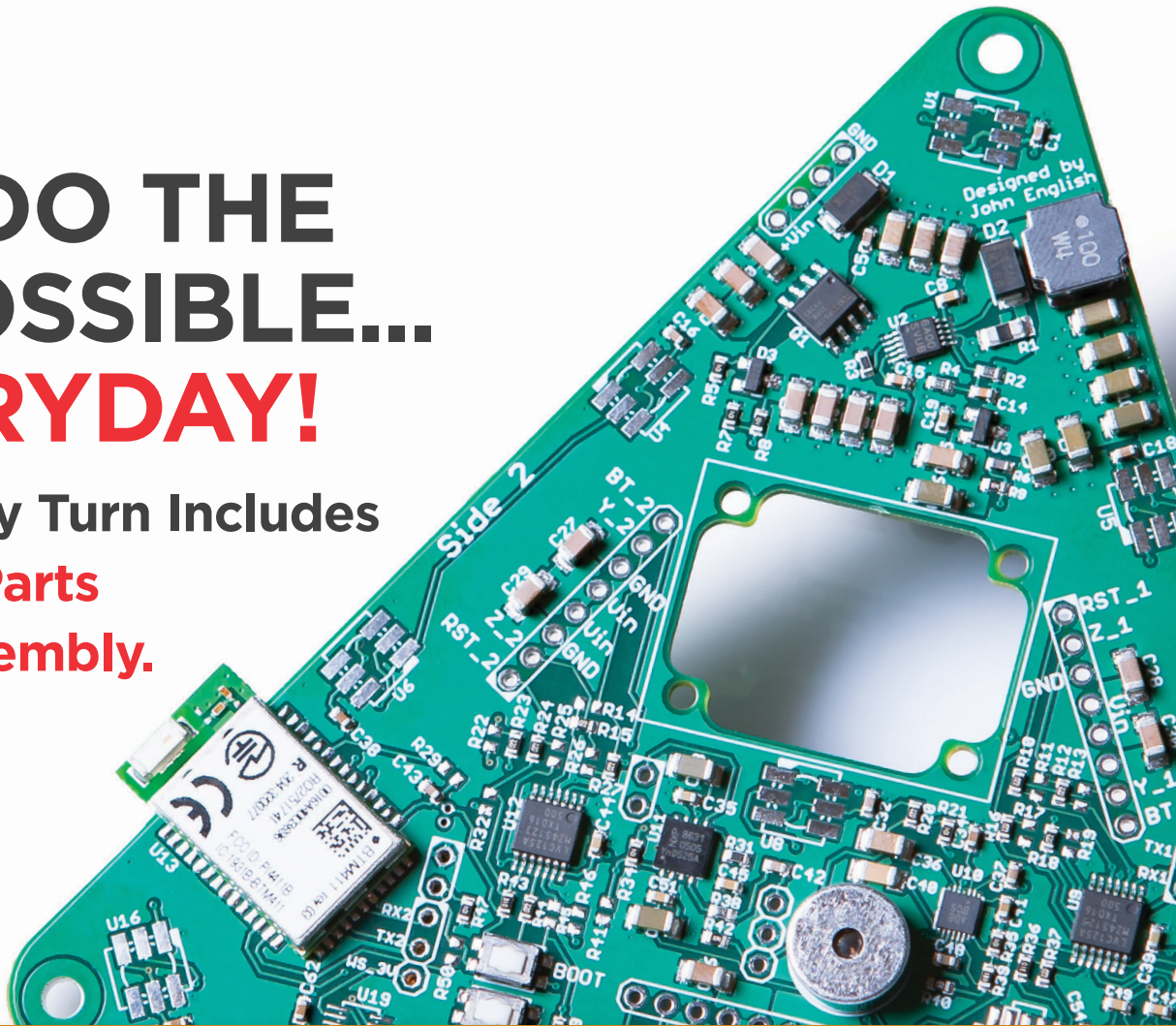
Espressif Systems | www.espressif.com

Microchip Technology | www.microchip.com

SparkFun | www.sparkfun.com

WE DO THE IMPOSSIBLE... EVERYDAY!

Our 5-Day Turn Includes
Boards, Parts
AND Assembly.



FREE LABOR

TRY SOMETHING DIFFERENT!

1st time customers receive FREE LABOR, up to \$1,000 on your first turn-key order.

OUR ASSEMBLIES START AT \$250

DOWNLOAD YOUR OFFER CODE HERE: Circuitcellar.com/SlingShot

We want to see **NEW DESIGNS** and **NEW CUSTOMERS!**

No more sacrificing quality for speed or price.

We are your **PCB ASSEMBLY SPECIALISTS!**

 **SlingShot**
ASSEMBLY



Find out why we're different at SlingShotAssembly.com/Different
Call for details: **720.778.2400** or Email: sales@sassembly.com

*Free labor, up to \$1000, for first-time customers on full turn-key assembly orders only.

for an NYSE stock symbol) had been input, the buffers reset, and an API call was made using that ticker. The keypad thread continuously checked if any input had been received from the keypad.

Reading and Sending via UART: API Call: Reading and Sending messages via UART between the PIC32 chip and the ESP Wi-Fi Module was tricky. To do these things, we used the function `DMA_PutSerialBuffer` provided by [3] and a heavily modified version of `GetSerialBuffer`. The `DMA_`

`PutSerialBuffer` function sent the string placed in `PT_send_buffer` through UART to the ESP module using DMA one byte at a time. Modifying the `GetSerialBuffer` function was tricky, because we could not find any documentation on how the ESP module responses were terminated. After experimenting, we concluded that most responses terminated with a '\n' (new line) and '\r' (carriage return) character in succession. We read up to 200 characters from the buffer (which was slightly more than the largest response we expected to receive), and stopped reading as soon as we saw that terminator. As each character was read, it was put into a character array that could be used by other functions. At the beginning of this function, we also cleared all UART2 errors.

Unfortunately, this method did not work for receiving stock price responses, which was the most important function of our project. To implement this task, we created a separate function, called `APICall()`, which was responsible for sending a custom command to the ESP via serial, based on the stock ticker entered on the keypad. The stock price was returned and displayed with its symbol and a triangle to indicate how the price instantaneously changed on the TFT LCD. To receive stock price responses, we read a preset number of characters from the buffer, instead of relying on a terminator. This was feasible because the response containing the stock price was always the same number of characters. We then parsed this response to get only the stock price, so we could display it and compare it with the last price.

RESULTS AND CONCLUSION

In conclusion, we finished with a reliable framework for making API calls to financial data servers. The latency for a call was roughly 1 s. The stock prices were updated every 10 s (manually set to avoid spamming the API services and reaching the daily call limit), provided we left the PIC running without requesting a different stock quote. Initial configuration of the chip took about 2 s, meaning our system had relatively small startup costs and could make quick and accurate updates in real time. Additionally, stock prices were reported to two decimal points accuracy.

Figure 4 shows some of the results. Overall, the results of the project met our expectations, despite various complexities along the way. In a broad sense, our project served as a proof of concept for lightweight wireless communication projects using Wi-Fi over the ESP module. Protothreads made our code more efficient, organized, and easier to understand.

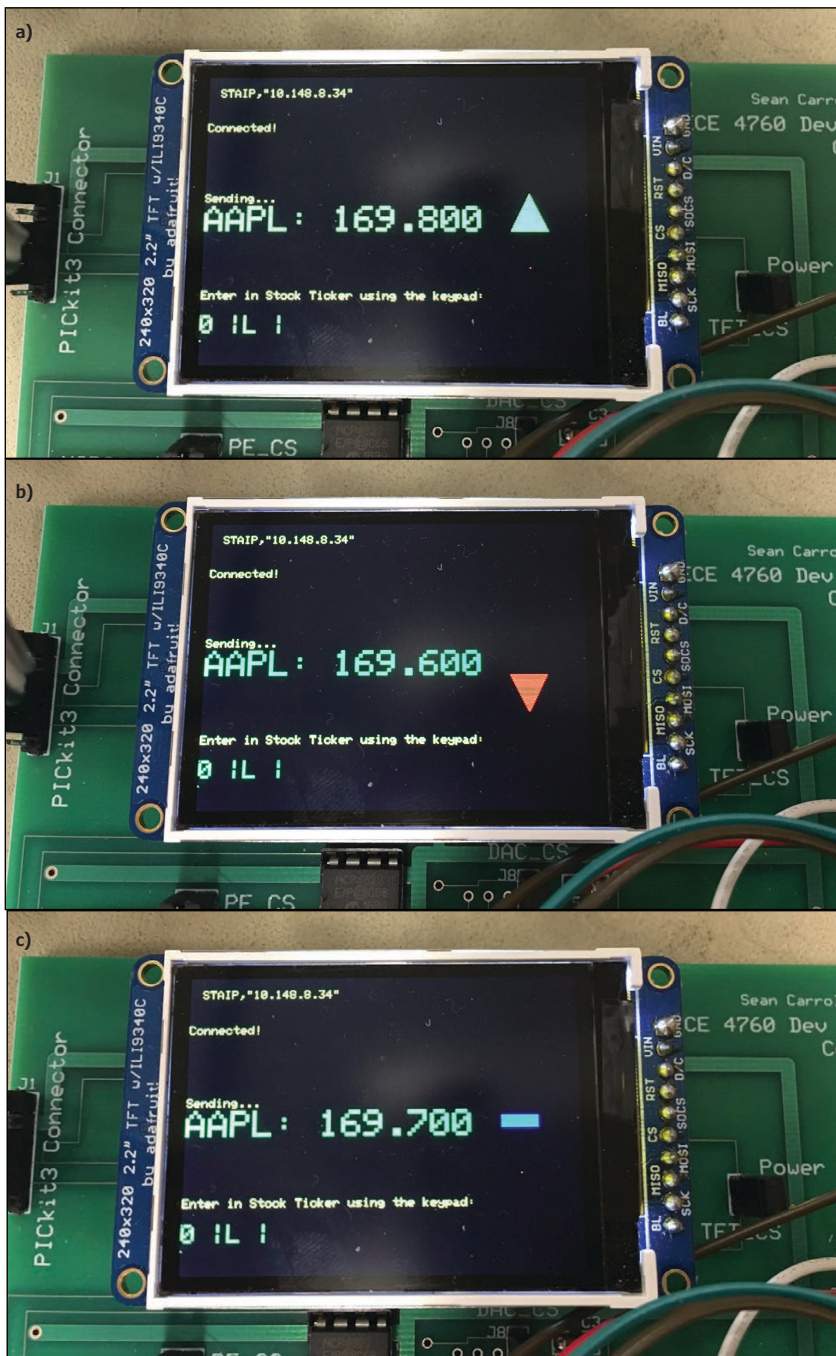


FIGURE 4

(a) Increase in price; (b) decrease in price; (c) no change in price

It is important to note the extensibility of the PIC32 MCU to other projects and applications. In general, this project served as demonstration of the PIC32 and the ESP as a TCP server and client. This manner of functionality could be easily applied to other peripheral circuits or additional modules. Wireless communication makes many new functions possible, and allows us to increase the usability and relevance of our smart home merely by incorporating additional API calls and data requests to various external servers.

In a similar fashion to our stock quote requests, we could make API calls to servers of the national weather service and display temperature, environmental conditions, and general weather data on our TFT display. We could take this even further by then analyzing the weather data and, based on the conditions, stream music that fit the mood of the weather. Streaming would occur via Wi-Fi communication with the ESP module.

Aside from the incorporation of API calls and data requests to external servers, Wi-Fi communication with the PIC could be used in various other applications. It is extremely versatile, so any sort of light display, temperature control circuit, home security unit, or IoT application could be implemented

using the ESP module and serial communication over the PIC. [E](#)

Authors' Note: Special thanks to our team member, Shrinidhi Kulkarni, for his contribution to the development of this project. Shrinidhi is a second-year masters student studying Applied and Engineering Physics at Cornell University. He helped develop some of the code for this project, and provided some reference text for this article.

ABOUT THE AUTHORS

A graduate of Cornell University, Saelig Khattar is currently a graduate student in Electrical Engineering at Stanford University and is a research assistant in the Stanford AI Lab. In his free time, Saelig enjoys playing tennis and eating soup dumplings.

David Valley graduated from Cornell last fall with a degree in Electrical and Computer Engineering. Some of his interests in electronics include embedded systems programming, PCB design and computer architecture. When he is not busy with classwork, he enjoys playing guitar, making playlists on Spotify and spending time outdoors. He will begin work as a software engineer this August.

Noritake Touch Works With:

Gloved Hand
Up to 9mm thick glass at 1mm air gap*
1.4mm Thick Leather Glove

Thick Overlay
Up to 15mm thick glass at 1mm air gap (bare hand)*

Air Gap
Up to 6mm air gap with 2mm thick glass (bare hand)*

*Noritake lab test results using GT800X480A-1303P

Touch TFT Modules

GT-1P Series (pictured)

- HID Compliant Touch
- DVI Video Input
- Plug-and-Play with PC/SBC

GT-CP Series

- Command-based Control

GT-EP Series

- Peripheral Control via iDevOS

Noritake itron
www.noritake-elec.com

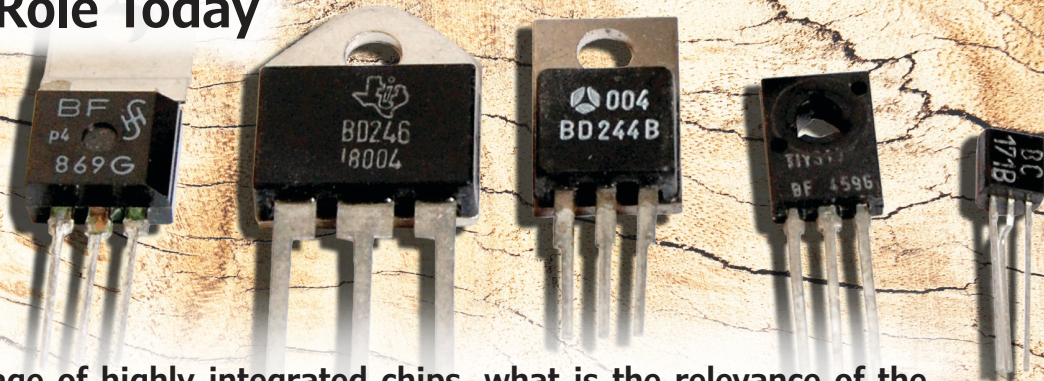
Phone: (847) 439-9020

marketing.ele@noritake.com

Transistor Basics

And Their Role Today

By *Stuart Ball*



In this day and age of highly integrated chips, what is the relevance of the lone, discrete transistor? It's true that most embedded system design needs can be met by chip-level solutions. But electronic component vendors do still make and sell individual transistors because there's still a market for them. In this article, Stuart reviews some important basics about transistors and how you can use them in your embedded system design.

What good is a transistor? Sure, integrated circuits (ICs) are full of transistors, thousands of them. Before the IC and microprocessor revolutions, there was a transistor revolution—where televisions, radios and computers were built using the new solid-state devices. The transistor was the father of the IC. But isn't a single transistor obsolete as a circuit element today? What use does a lowly transistor have in a world where the current Intel microprocessors have over a billion transistors each?

It's true that nearly all the things we used to do with transistors can be done cheaper, better and more efficiently with an IC, and we can do things with ICs that are not possible with discrete transistors. It would not be possible to build a modern microprocessor

with discrete transistors—the lead lengths alone would make the speeds impossible. But the reverse is also true. A discrete transistor can be a simple way to solve some problems. Transistors, for example, typically have much higher operating voltage and power limits in simple circuits than those of comparable ICs. Electronics manufacturers and distributors still make and sell individual transistors because the parts still have some uses. In this article, I want to go over some very basic things about transistors, how they are used and how you can include them in your applications.

OVERVIEW

A BJT (bipolar junction transistor) was the first commonly available transistor, and it fueled the transition away from vacuum tubes. BJTs come in two varieties, NPN and PNP. Both are (usually) silicon devices. The silicon is modified (doped) with impurities to produce N-type or P-type material. An NPN transistor has a P-type layer sandwiched between two N-type layers, and a PNP is the reverse.

Figure 1 shows an NPN BJT schematic symbol, a simple diagram of the structure, and a diode model. The N-P-N structure is just representative. In an actual transistor, the collector region is normally larger than the emitter region, and none of them is square as shown in the diagram. The diode representation of the transistor indicates how current flows, not how the actual part is constructed. You can't build a transistor out of two diodes, but using two diodes helps to explain how the transistor biasing works.

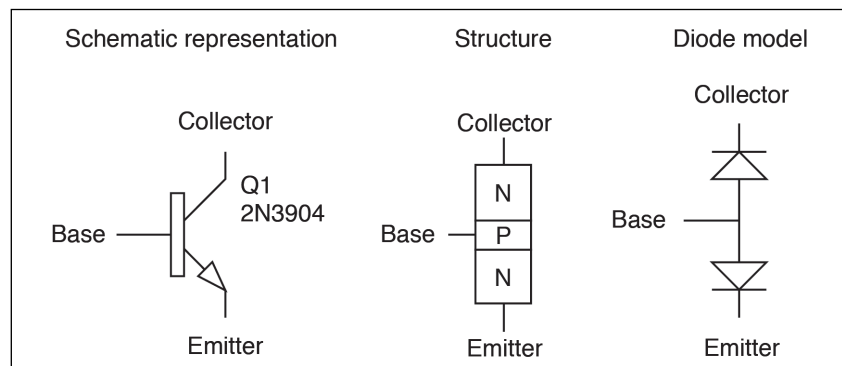


FIGURE 1
Schematic symbol, physical representation and diode model of NPN transistor

Operation of an NPN transistor is conceptually easy to understand. Referring to the diode model, if you connect the collector to a positive voltage—say 5 V—and the emitter to ground, you end up with two diodes back-to-back with their anodes connected together. The junction of the two anodes represents the base of a transistor. If you apply a positive voltage greater than 0.7 V to the base, the emitter diode will be forward-biased and current will flow from the base, through the emitter and to ground. The collector diode will be reverse-biased, and no current will flow through that diode.

REAL TRANSISTOR OPERATION

Now discard the diode model and look at a real transistor. If the collector is connected to +5 V and the emitter to ground, and the voltage on the base is high enough (0.7 V) to forward-bias the base-emitter junction, current will flow from the base to the emitter *and* from the collector to the emitter. If the base-emitter voltage is below 0.7 V, the transistor is in “cutoff” and no current flows through the emitter or through the collector. That’s it. That’s how a BJT works.

The collector-emitter current flow is inherent in the construction of the transistor. It’s why the actual transistor differs from the diode model, and it’s why you can’t build a transistor from two diodes. If the collector is at +5 V and the emitter is at ground, bringing the base to about 0.7 V will cause current to flow from the 5 V supply—through the collector—to the emitter and to ground. If the emitter is at +2 V, then you must bring the base to about 2.7 V to get current to flow from the collector to the emitter.

The magic in a transistor is determining how to get the amount of current you want flowing through the collector. If you just connect the transistor as I’ve described, with nothing to limit the current, your transistor will quickly become a smoking, melted bit of plastic.

Generally, if the transistor is operated within its current, power and voltage ratings, the current in the emitter will be the current flowing into the base plus the current flowing from the collector to the emitter. A very small base current controls a much larger collector current, so the collector current is approximately equal to the emitter current. When no current is flowing in the collector, the transistor is in “cutoff” as mentioned earlier. If there is enough current flowing that the collector-emitter voltage is as low as it can go (generally around 0.3 V for a small-signal transistor), the transistor is considered “saturated”. In this state, changes to base current no longer affect collector current.

PUTTING IT TO USE

How might we use this transistor? **Figure 2** shows a simple circuit. In this circuit, we connect

the collector to +5 V, the emitter to ground through a 220 Ω resistor and the base to a fixed value of 1 V. The forward voltage of the 2N3904 is 0.65 V to 0.85 V at 10 mA collector current. Conventionally, 0.7 V is used for calculations. So, the voltage at the emitter (VE) will be 1 V - 0.7 V, or 0.3 V. Here’s where the magic happens: The voltage at the emitter is fixed, so the current through the 220 Ω resistor is $0.3\text{V}/220\Omega$, or 1.36 mA. The collector current is the same. Therefore, by controlling the base voltage, we control the emitter current and thereby the collector current.

Figure 3 shows how we can make an amplifier with this circuit. This circuit is identical to the circuit in Figure 2, except that now we’ve added a 1.5 k Ω resistor, R2, between the collector and the 5 V supply. Since the current in the emitter is fixed at 1.36 mA, the current in the collector is also 1.36 mA. This current flows through R2, producing a voltage across R2 of $1.36\text{ mA} \times 1.5\text{ k}\Omega$, or 2.04 V. So, the voltage at the collector, VC, is the 5 V supply minus the voltage across R2, or 2.95 V.

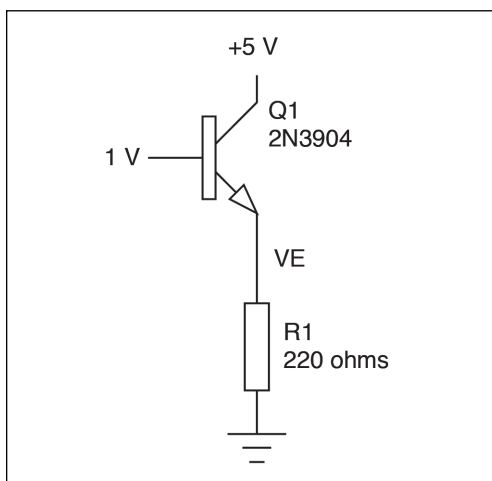


FIGURE 2

A simple circuit shows the base-emitter voltage and current relationship

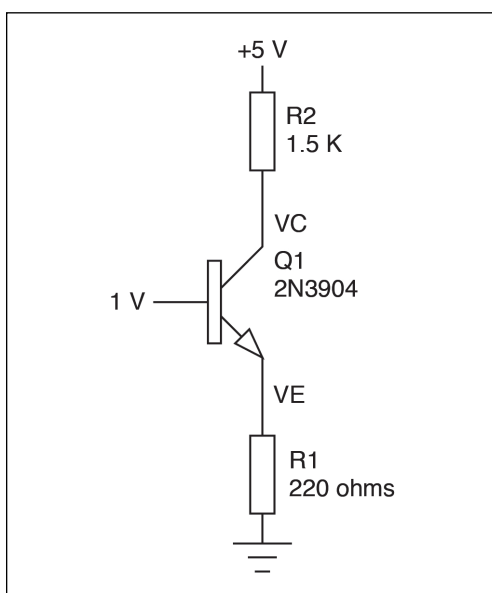


FIGURE 3

The transistor connected as an amplifier by adding a resistor in the collector

Now, what happens if the voltage at the base is raised to 1.1 V? When that happens, the voltage at the emitter is now 0.4 V ($1.1 \text{ V} - 0.7 \text{ V}$), making the emitter current 1.8 mA. The collector current is also 1.8 mA, so the voltage across R2 is now $1.8 \text{ mA} \times 1.5 \text{ k}\Omega$, or 2.7 V. VC is now $5 \text{ V} - 2.7 \text{ V}$, or 2.27 V. So, a 0.1 V change in the base voltage caused the collector voltage to drop from 2.95 V to 2.27 V, a change of -0.68 V. The collector voltage dropped by $6.8 \times 0.1 \text{ V}$ (the input voltage change).

Here's the interesting thing: The collector voltage change is equal to the negative of the input voltage change times the ratio of the collector resistor R2 to the emitter resistor R1, or $1.5 \text{ k}\Omega / 220 = 6.8$. If you work through the math, this makes sense, because the collector current is the same as the emitter current. But since the collector resistor R2 is 6.8x the emitter resistor, any current change in the emitter resistor will result in a voltage change 6.8x as large at the collector.

If you did the same calculation after lowering the base voltage from 1 V to 0.9 V, you would see the collector voltage rise by 0.68 V. This circuit is an inverting amplifier

with a gain of -6.8. A positive voltage change at the input produces a negative voltage change at the output and vice-versa.

This circuit has some limitations. If you put 1.32 V at the base, you will find that the emitter is at 0.62 V, and the collector voltage works out to be nearly the emitter voltage. The transistor can't drive the collector to the emitter voltage, so it's saturated. The limitation of this specific circuit, therefore, is a maximum input voltage of about 1.3 V. At the other end, anything less than 0.7 V causes the transistor to go into cutoff. So, the useful input voltage range of this circuit is 0.7 V to about 1.3 V. Still, that would be adequate for boosting a low-level audio signal to something that can be further amplified.

Speaking of audio, how would you connect audio signals into the circuit? Audio signals typically swing between negative and positive voltages. If you put that into the base, the transistor will be in cutoff most of the time—all the time if the positive signal peaks never reach 0.7 V.

This brings us to biasing. **Figure 4** is a modification of Figure 3 with some biasing resistors added to the base. Resistors R3 and R4 make a voltage divider that brings the base to about 1 V. This is halfway between the 0.7 V and 1.3 V lower and upper limits of the circuit. Now say that we apply a signal to the input that swings between -0.1 V and +0.1 V. Because of the DC blocking capacitor C1, this will become 0.9 V to 1.1 V at the base, and will be amplified by -6.8 in the circuit.

There are other ways to bias a transistor base. A voltage reference diode, as shown in **Figure 5**, fixes the base at a known voltage. In this circuit, the emitter voltage, VE, will be about 1.3 V, so the emitter and collector current will be 5.9 mA. The point is not to show all the possible ways to bias a transistor, just that there are other ways to do it.

TRANSISTOR LIMITATIONS

As with all things in the physical world, transistors have some limitations. We already looked at one—the values of the base and emitter resistors in the amplifier circuit have to be chosen so that the transistor doesn't go into cutoff or saturation with whatever input signal you are trying to amplify.

Transistors have other characteristics. For example, the 2N3904 used in these examples has a maximum collector-emitter voltage of 40 V. Any more than that, and the transistor fries. The base-emitter reverse voltage—where the base is taken negative with respect to the emitter—has a maximum value of 6 V. Beyond that, the emitter-base junction breaks down.

The collector can handle a maximum continuous current of 200 mA. The device

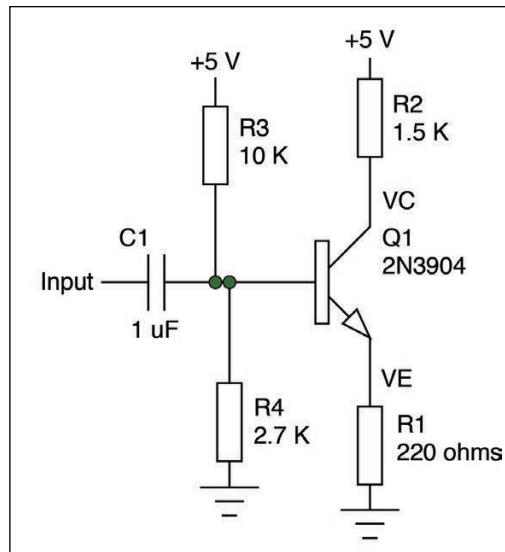


FIGURE 4

Biasing resistors allow the transistor to operate with AC-coupled inputs such as audio signals.

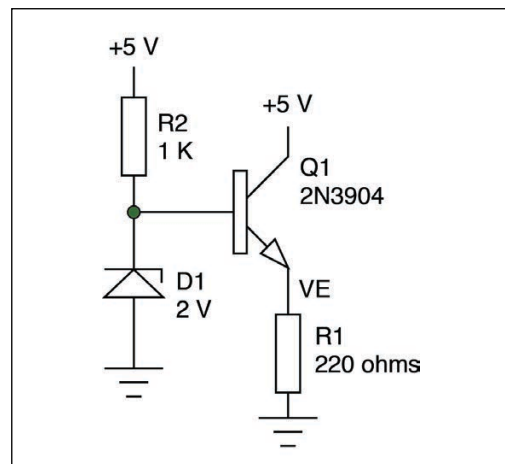


FIGURE 5

A Zener or reference diode can be used to create a fixed bias.

has a maximum power dissipation of about 600 mW. So even though the collector-emitter can withstand 40 V and the collector current can be as high as 200 mA, if you try to put 200 mA through it at 40 V, it will fail. 40 V at 200 mA is 8 W, well beyond the power-handling capability of the device.

The point of all this is that, like any semiconductor device, your design has to stay within all the maximum ratings: Power, collector-emitter voltage, collector current, emitter-base reverse breakdown voltage and so on.

One of the key characteristics of the transistor is the current gain. This number describes how much the emitter current changes for a given change in the base current. The current gain varies with the amount of current flowing in the collector. For the 2N3904, the minimum current gain at 0.1 mA collector current is 40. At 10 mA, the minimum gain is 50. The maximum gain per the datasheet is 300. Just before writing this paragraph, I measured a handful of 2N3904s. All of them had gain exceeding 300.

The practical implication of the gain is to affect how the emitter interacts with the base. If the transistor in the amplifier circuit in Figure 3 had a gain of only 10, the 220 Ω resistor in the emitter would look like approximately 2 k Ω at the base, which would affect biasing and the load presented to the driving circuit. In that case, you would want the biasing resistors to be a low enough value that the loading effect of the emitter resistor would change the bias voltage by less than 10% or so. But if you have to use lower value resistors in your biasing circuit, this in turn presents more load to whatever is driving it. In the case of the amplifier, it reduces the overall end-to-end gain.

Fortunately, for most small-signal applications, it isn't too hard to find a transistor with a sufficiently high minimum gain to make this a minor problem. Where you get into difficulty is when you need a very low value of emitter resistance. Even at a gain of 300, an emitter resistor of about 10 Ω could have a significant loading effect on the base that must be considered in your calculations. Because the transistor has finite gain, you can't use very large resistors—such as something in the megaohm range—to bias the base. If you do, the emitter will pull down the voltage.

One common addition to an audio amplifier is to bypass the emitter resistor with an electrolytic capacitor. The capacitor has a very high impedance (nearly infinite) at DC, but the impedance decreases as frequency increases. This allows the DC biasing to work, but it raises the gain for audio signals by making the emitter impedance (the resistance in parallel with the impedance of the capacitor) a very low value at audio frequencies. This makes the ratio of the collector resistance to emitter resistance much higher at audio than at DC, which raises the gain. (Remember: The gain is the collector resistor divided by the emitter impedance.) However, this also has the effect of significantly lowering the input impedance of the circuit at those audio frequencies. Other transistor characteristics that affect use in RF circuits, such as high-speed switching circuits, are beyond the scope of this article, and won't be discussed here.

APPLICATIONS

You can build amplifiers with transistors, and a lot of people do. But it's also easy to build an amplifier with an op amp or other IC and I want to focus here on applications where the unique characteristics of a transistor are useful.

How might you make practical use of a transistor, given what we've done so far? In **Figure 6**, I have modified Figure 5 by

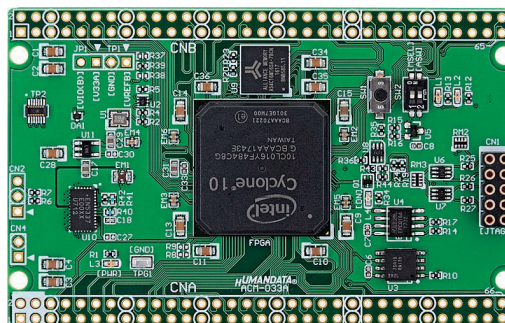
FPGA Boards from HUMAN DATA

SAVING COST=TIME with readily available FPGA boards

- Basic and simple features, single power supply operation
- Free download technical documents before purchasing

INTEL ACM-033

Intel Cyclone 10 LP F484 FPGA board



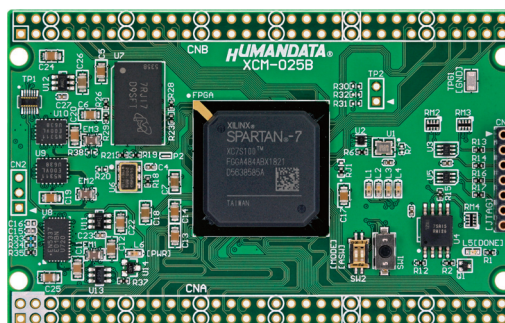
Cyclone 10 LP SDRAM SPI-Flash RoHS

SIZE : 3.386" x 2.126" (86 x 54 mm)

ACM-033 is an FPGA board with Intel high-performance FPGA Cyclone 10 LP. It's compact and very simple. 3.3V single power supply operation.

XILINX XCM-025

Xilinx Spartan-7 FGGA484 FPGA board



Spartan-7 DDR3 RoHS

SIZE : 3.386" x 2.126" (86 x 54 mm)

XCM-025 is an FPGA board with Xilinx high-performance FPGA Spartan-7. It's compact and very simple. 3.3V single power supply operation.

See all our products, A/D D/A conversion board, boards with USB chip from FTDI and accessories at :

www2.hdl.co.jp/CC19B



making the reference voltage 2.5 V, making R1 120 Ω and adding an LED in the collector circuit. Because the voltage at the base is fixed at 2.5 V by the reference diode, the emitter voltage is 1.8 V and the emitter current is 15 mA. This is true as long as the V+ supply voltage is high enough to keep the reference diode and LED turned on. So, the LED will have 15 mA current whether the supply voltage is 5 V or 20 V.

Obviously, there are upper limits to this, and at some point, the voltage or power dissipation limit of the 2N3904 will be exceeded and it will go up in a cloud of smoke. I've shown the bias circuit powered from 5 V. If you also powered it from the variable V+, you also would need to take the limitations of R3 and D1 into account. But if you wanted a constant current through an LED regardless of supply voltage (within reasonable limits), this circuit will do it. You might do this if you wanted an LED to have constant intensity regardless of the voltage applied, or just to keep higher voltages from exceeding the maximum LED current.

Figure 7 shows a 2N3904 used for logic-level translation between two different circuits operating at different voltages. You might use

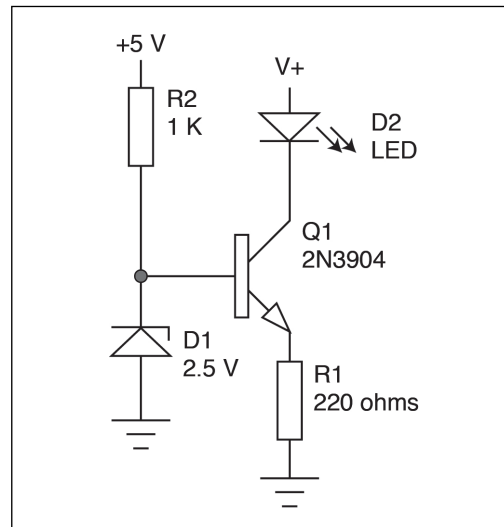


FIGURE 6

A 2N3904 connected as a constant-current LED driver

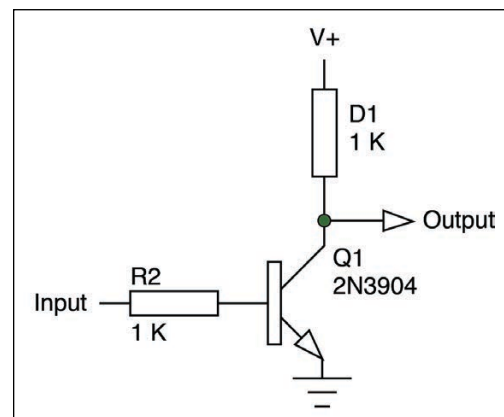


FIGURE 7

A 2N3904 used as a logic-level translator

this to translate between a 3.3 V output of a microcontroller (MCU) to the input of a circuit that needs 5 V. V+ in the schematic would be connected to the supply voltage of the target system. Whatever is driving the input must have enough output current capability to drive the 2.2 k Ω resistor. This circuit inverts the signal—a high input produces a low output. In this circuit, the transistor is always in either cutoff or saturation.

There are plenty of ICs that can do this, such as open-collector buffers, so why use a transistor? The transistor can handle higher voltages than most logic-level translator circuits. A transistor could translate between a 3.3 V circuit and a 12 V circuit, for example.

Many voltage-translator circuits require that you know the supply voltage, and therefore the drive voltage, of the input. But I had a situation once where the input could come from different sources, ranging from under 2.5 V to 5 V. The transistor solution works for all logic voltages, because the transistor will turn on with any drive voltage above 0.7 V. It could even be used to translate between a 12 V or 24 V input to a 3.3 V or 5 V output, as long as the input resistor R2 is large enough to prevent excessive current.

The final NPN application is shown in **Figure 8**. In Figure 8a, a 2N3904 is driving a relay. The diode D1 protects the transistor against overvoltage. When the relay is turned off by switching the transistor off, a “flyback” voltage is created as the energy in the relay coil is dissipated. This voltage can reach levels sufficient to destroy the transistor due to excessive collector-emitter voltage—remember the transistor characteristics section. Diode D1 limits the voltage to 0.7 V above V+ to protect the transistor. But this has the side effect of slowing down relay opening.

Figure 8b shows the same circuit, but with a 12 V Zener, D2, in series with D1. This allows the flyback voltage to reach 12.7 V above V+, which allows the coil energy to be dissipated much more quickly, speeding up relay operation. But with a 12 V relay, the collector voltage will exceed 24 V during the flyback period. This circuit takes advantage of the high collector-emitter breakdown voltage to improve the speed. There are some relay drivers that can do this, but they offer little advantage over a transistor. Note however that base resistor R1 must be sized to allow enough current for the transistor to operate the relay. A large, high-current relay may require a pre-driver and a power transistor. At that point, an IC might be a better solution.

PNP TRANSISTORS

I've focused on NPN transistors so far. Functionally, the PNP is the reverse of the

NPN. The collector voltage of the PNP (when normally biased) is less than the emitter, and the base is lower than the emitter by 0.7 V to turn the transistor on. It isn't necessary to use negative voltages. As with the NPN, the voltage with respect to the emitter is what matters. A PNP transistor can be paired with an NPN in simple audio amplifiers to make a headphone or speaker amplifier. The PNP complement to the 2N3904 is the 2N3906.

Figure 9 shows how a 2N3906 might be used to create a negative bias voltage in a system with only a positive supply. You might need a negative bias to offset an input signal, or to power an op-amp that needs a negative supply for some reason.

The input is driven by a square wave input that might come from the timer output of a MCU or a two-transistor multivibrator (Google it). I picked values arbitrarily for the components in this example. You would want to use component values appropriate for the input frequency, output current and voltage, and other requirements of your application. Note that the input signal must swing close to the positive supply rail (5 V in the circuit shown) to fully turn off Q1—otherwise the transistor will never turn off, and it will get hot. If you were driving the circuit with a logic-level output, you might need a pull-up resistor to be sure the input swings all the way to the positive rail. You could also use this circuit in a 3.3 V system.

I include this example to show how a PNP transistor can be used. This isn't to say there aren't ICs that can do this. For example, the TPS6735 DC/DC converter made by Texas Instruments can produce a -5 V output at 200 mA, although it won't operate at 3.3 V.

MOSFET TRANSISTORS

I've looked at BJTs so far, but there is another class of transistors called MOSFETs (metal-oxide semiconductor field effect transistors). Where a BJT has a base, emitter

ABOUT THE AUTHOR

Stuart Ball is a registered professional engineer with a BSEE and an MBA. He has more than 30 years of experience in electronics design. He is currently a principal engineer at Seagate Technologies.

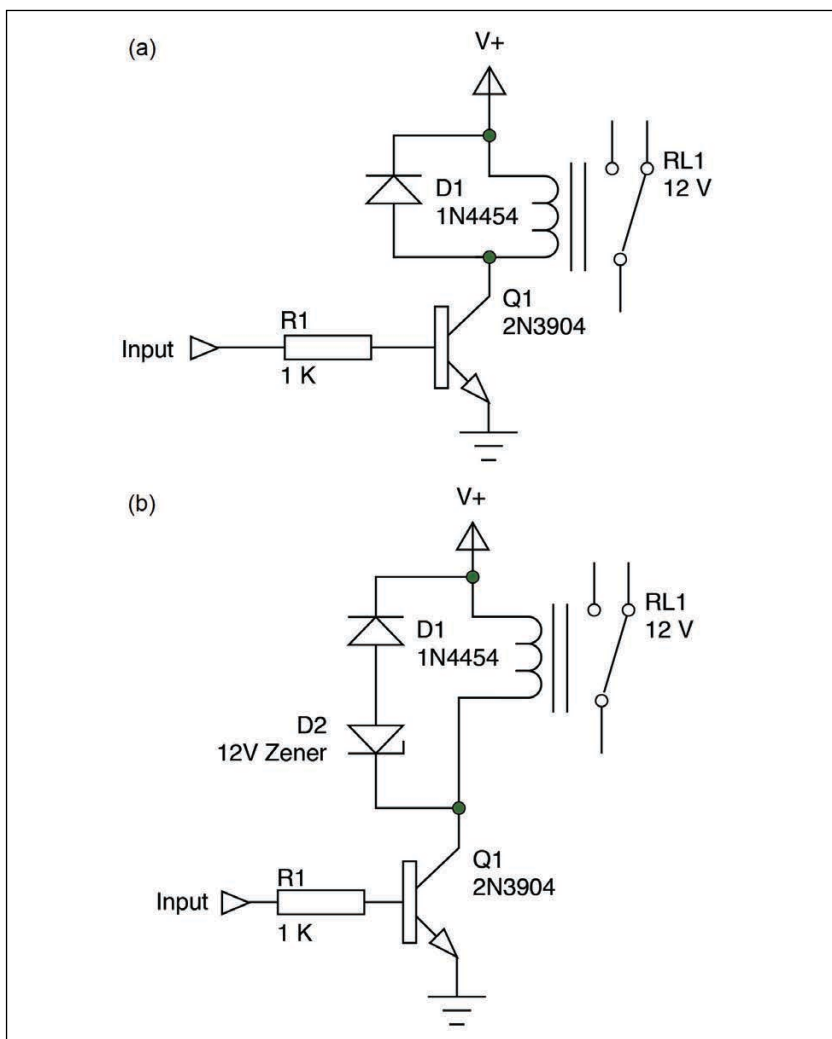


FIGURE 8 Driving a relay with a 2N3904. A basic diode clamp (a) and a higher voltage Zener clamp (b) for faster operation.

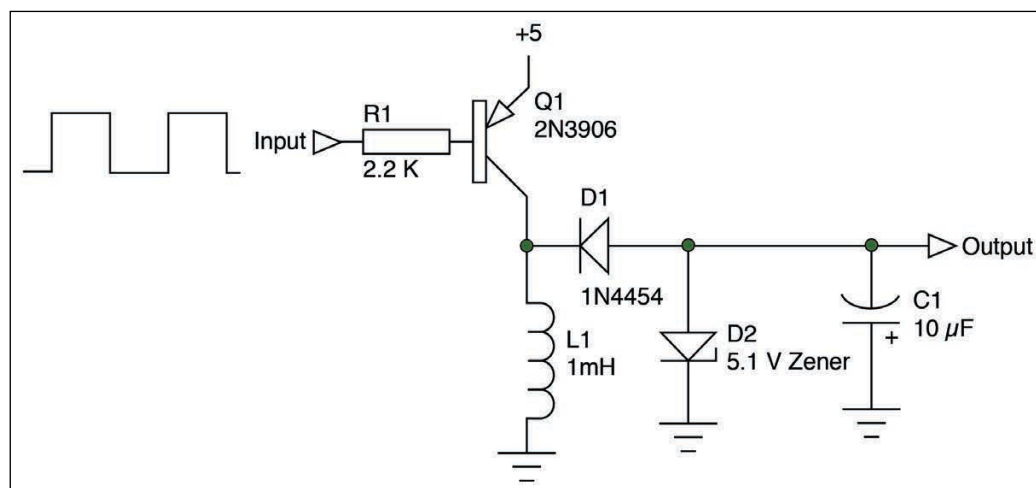
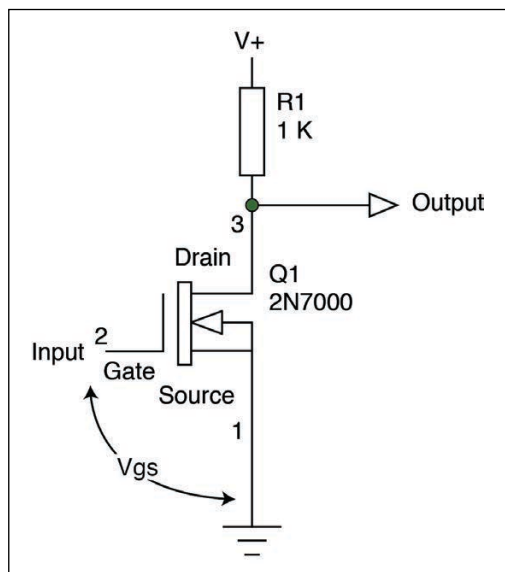


FIGURE 9 Negative voltage generator using PNP 2N3906

FIGURE 10

2N7000 MOSFET as inverting logic-level converter



and collector, the equivalent MOSFET pins are the gate, source and drain. MOSFET operation is similar to the BJT, but there are some important differences.

The MOSFET was sometimes previously referred to as the IGFET (insulated-gate field effect transistor). I haven't seen that term used for many years, but it is descriptive. The gate of the MOSFET is electrically insulated from the rest of the part, and the current from the drain to the source is controlled by the electrical field created by applying a voltage to the gate. The insulated gate means that the MOSFET has a very high impedance input, so no current has to flow into the gate to control the drain-source current. In fact, if current is flowing into the gate, it probably means that some limit has been exceeded and the transistor has failed.


The BJT can be thought of as a current-controlled current device, where a small change in base current causes a large change in collector current. A MOSFET is a voltage-controlled current device, where a change in the gate voltage causes a large change in the drain current. **Figure 10** shows a 2N7000 MOSFET connected as a logic-level translator, similar to the way the BJT was wired in Figure 7. It will work the same way as the 2N3904 circuit, with the following differences:

1. The high impedance means no series resistor is needed in the gate to limit current. This also means that the transistor input won't load down whatever output is driving it.
2. The BJT needs 0.7 V and a little current to turn the transistor on. The MOSFET needs the gate to be positive with respect to the source. In the case of the 2N7000, the turn-on voltage, V_{gs} , can range from 0.8 V to 3 V. This means that using a 2N7000 to translate between a 2.5 V or 3.3 V input to a higher voltage output might be problematic, and the transistor might not turn on. However, going the other way, from a 5 V or higher system input to 3.3 V or 2.5 V output, will work the same as it does with the bipolar circuit.
3. A saturated MOSFET doesn't have a saturation voltage—it has a resistance between the source and drain. For the 2N7000, this can be up to about 6 Ω when $V+$ is 5 V for the On Semiconductor version of the part. For most applications, this value is small enough that it makes no difference, but it is something to be aware of, especially when switching significant current.

The 2N7000 is normally used as a switch. You can bias it as an amplifier, but the varying V_{gs} threshold value makes that a bit more complicated than for a BJT. Like the PNP complement to the NPN transistor, N-channel MOSFETs have a complement, which is the P-channel MOSFET. The BS250 from Vishay is an approximate P-channel equivalent to the 2N7000. You could use such a transistor instead of a PNP to implement the negative voltage generator mentioned earlier, although, of course, you have to be sure the driving voltage exceeds the gate threshold voltage.

OTHER TRANSISTORS

I've focused on small-signal transistors to demonstrate the basic principles. In both bipolar and MOSFET transistors there are devices designed to handle high currents and high voltages, parts designed specifically for RF applications, and other variants. But the basic principles are the same.

I hope my explanation of how transistors work has helped you understand them better, and that the examples are enough to let you experiment with transistors in your applications. Sometimes transistors are useful, even though they've been around a long time. And even in circuits you could build with ICs, transistors are interesting devices for tinkering, because you can get down to the basic component level. 

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

RESOURCES

On Semiconductor | www.onsemi.com

Texas Instruments | www.ti.com

Vishay | www.vishay.com

ROBO Business

October 1-3, 2019

Santa Clara Convention Center, Santa Clara, CA

SAVE 20%

Use Code
CCRB19

INNOVATORS WELCOME

See the newest and most innovative technologies in the robotics industry. From sensors and LiDAR to grippers and motors to mobile robots and cobot arms, RoboBusiness brings together every corner of the industry to help you succeed.

NEW for 2019

- 2 Innovation & Technology Tracks
- Expanded networking opportunities
- The Emerging Markets Forum
- Tech Talks at the Expo Theater

Robo Favorites Return

- 7th Annual Pitchfire Start-up Competition
- VC Office Hours at the Start-Up Incubator
- Robotic demos on the expo floor

Don't Miss These:

- Innovative Keynotes
- Educational Breakouts
- New Technologies on the Expo Floor

REGISTER TODAY! ■ RoboBusiness.com

Robotic Arm Plays Beer Pong

Using PIC32s and IMUs

By *Daniel Fayad, Justin Choi and Harrison Hyundong Chang*

Simulating human body motion is a key concept in robotics development. With that in mind, learn how these three Cornell graduates accurately simulate the movement of a human arm on a small-sized robotic arm. The Microchip PIC32 MCU-based system enables the motion-controlled, 3-DoF robotic arm to take a user's throwing motion as a reference to its own throw. In this way, they created a robotic arm that can throw a ping pong ball and thus play beer pong.

Because the electronics industry is advancing so rapidly, it's now easier than ever to build interesting systems without hurting your wallet. This has also led to an overwhelming variety of projects suitable as a final project in a college class. One of the types of systems we were interested in from the start was wearables—but wearables and what? We could attempt to create something meaningful, something that would help humanity. Or not. We opted for a something fun for college students—something you could talk about at parties without losing everyone's attention. That's how our idea of the "Pong Bot" came to life.

The project's focus was to simulate the movement of a human arm such as aiming and throwing small objects—for example, a ping pong ball—with a small robotic arm. We used the motion-controlled, 3-DoF (degrees of freedom) robotic arm that takes the user's throwing motion as a reference to its own throw. A robotic arm that mimics the user's arm motion has many different applications. For example, you can use the robot arm to lift heavy objects that human arms can't handle, or use the robotic arm remotely from a distance.

With all that in mind, we hope that, with small modifications, readers could take the concepts from this project and create something more

useful—although perhaps less fun. Integrating both mechanical and electrical components, we set out to control a beer-pong catapult robot that simulates the user's throwing gesture. With this system, a mini-scale beer pong game can be played using a robotic arm that throws the ping pong ball for you—a fun twist enabling you to play beer pong in style. For those unfamiliar with the game, beer pong is a drinking game in which players throw a ping pong ball across a table with the intent of landing the ball in a cup of beer on the other end.

THE USER INTERFACE

The user interface for our device relies on a sleeve (**Figure 1**) worn on the user's arm and adjusted so that IMUs (inertial measurement units) align with wrist and elbow. This allows gesture control. The aiming of the catapult and start position for the throw are determined using readings from the IMU. The IMU delivers two angles from the elbow and one angle from the wrist, so we get the 3 degrees of freedom needed for our robotic arm. By combining the data from the gyroscopes and accelerometers attached on the elbow and wrist, the controller sends out three current angles from the calibrated zero. This mapping of user's arm to the robotic arm will be discussed in a later section including the implementation of a



complementary filter using IMU readings.

The controller device itself consists of two IMUs, a Digi International RF XBee module, an inexpensive pressure sensor and a Microchip PIC32 microcontroller (MCU) on the sleeve. The pressure sensor is implemented as a digital button. When the user pinches on the pressure sensor to a certain threshold, the robotic catapult starts to move according to the movement of user's arm. This is when the user is expected to aim. As soon as the pressure sensor is released, the robotic arm swings very quickly as it throws the ball—similar to what a catapult would do. **Figure 2** shows the development board and the circuitry mounted on the sleeve. The development board used is Sean Carroll's PIC32 Small Development Board—a link with the details of this resource is provided on the *Circuit Cellar* article materials webpage.

The release mechanism was designed after initial testing. Due to a short delay between the user's movement and the movement of the servos, it was difficult to get a rapid movement that would cause the ball to be thrown. For maximum enjoyment the user is encouraged to make the throwing movement after the pressure sensor has been released. This won't affect the throw, but it makes it seem more like the user is also controlling the throw.

The controller can wirelessly communicate to the robotic arm by having all the sensors on the arm hooked up to a local PIC32 MCU that sends signals via an RF transmitter. On the robotic arm, we will have an RF receiver that receives the signals from the controller and moves the robotic arm accordingly in real-time. Because of what we believe to be hardware constraints with the Xbee modules, we were only able to send a signal from the controller to the robotic arm every 200 ms. This means that the servo's position signal was updated every 200 ms. This sometimes made the robotic arm seem a little shaky and unresponsive. For a more reliable system, our final version didn't use wireless communication. Instead, we used two very long wires to directly connect UART pins between the two boards. This modified, final version enabled us to establish a stable and fast communication interval of 65 ms, providing smooth control of the robotic arm. The long wires won't interfere with the user's movement as long as the arm is used as intended.

ROBOTIC ARM

Due to time and budget constraints, we had to get creative with our materials and assembly. Most of the materials were collected while we were in a cafeteria during a break from working on this. The assembled robotic arm is shown in **Figure 3**. Despite the commonplace materials, it behaves just as we intended, and serves its purpose. The system requires three servos to

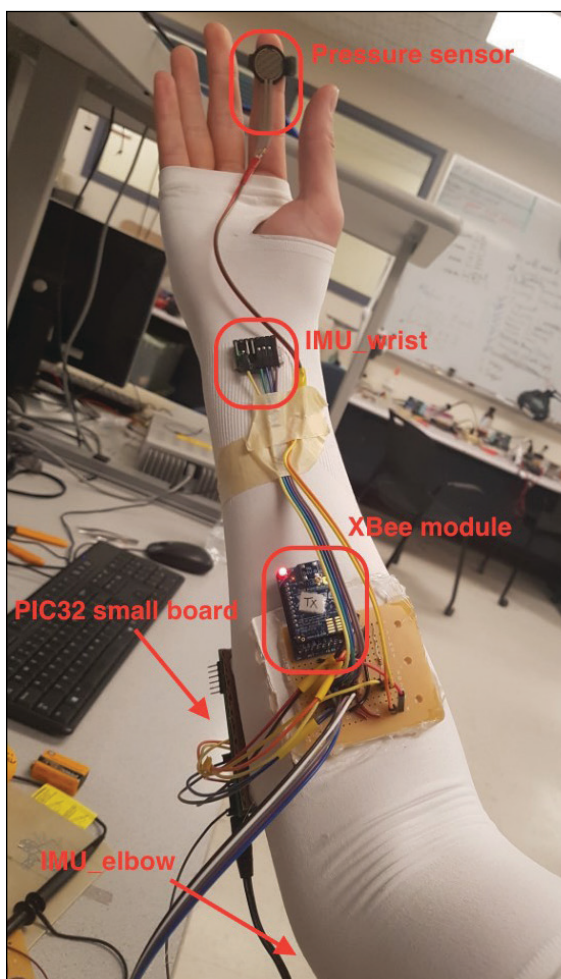


FIGURE 1 Shown here is the sleeve as worn by the user. The IMU_elbow is on the other side and cannot be seen in this figure. The sleeve is elastic, so the user can adjust the location of the IMUs for calibrations.

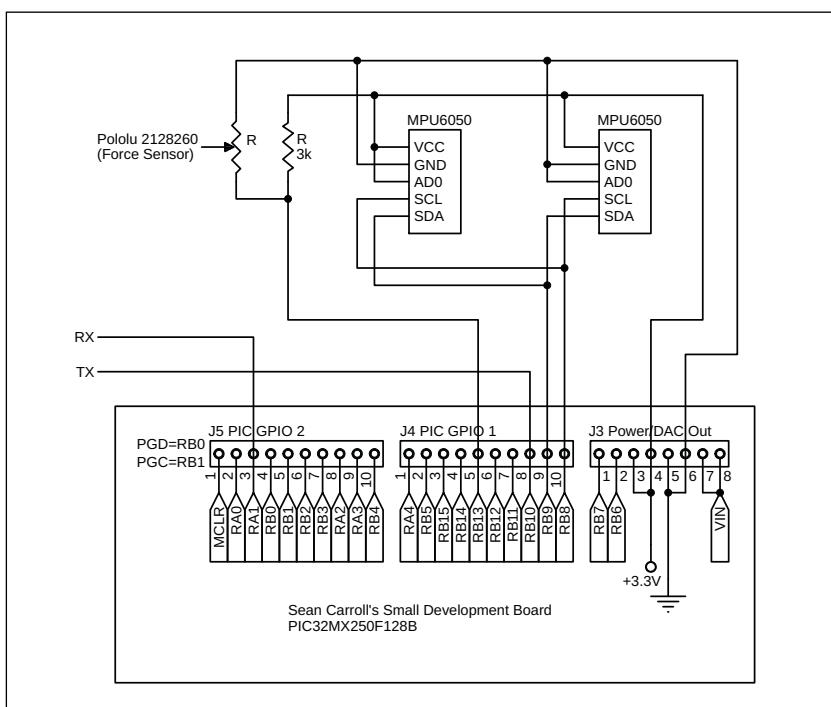


FIGURE 2 This is the schematic of the development board and the circuitry mounted on the sleeve shown in Figure 1. The XBee modules are ignored because they weren't part of the final demo. Accordingly, we just show the UART connections as RX and TX that are connected to the second PIC32 MCU (shown in Figure 5) [1].

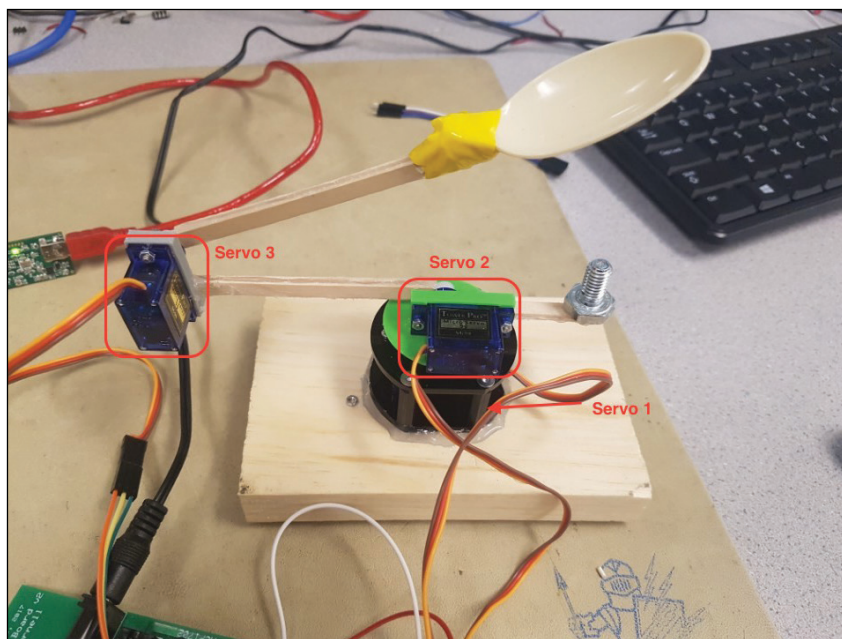


FIGURE 3

The robotic arm is made of coffee sticks, small servos and a spoon—held together by hot glue and tape. The screw seen on the opposite side of Servo3 served as counterweight, because the weight of Servo3 combined with the quick movements of Servo2 sometimes made the arm detach from the servo.

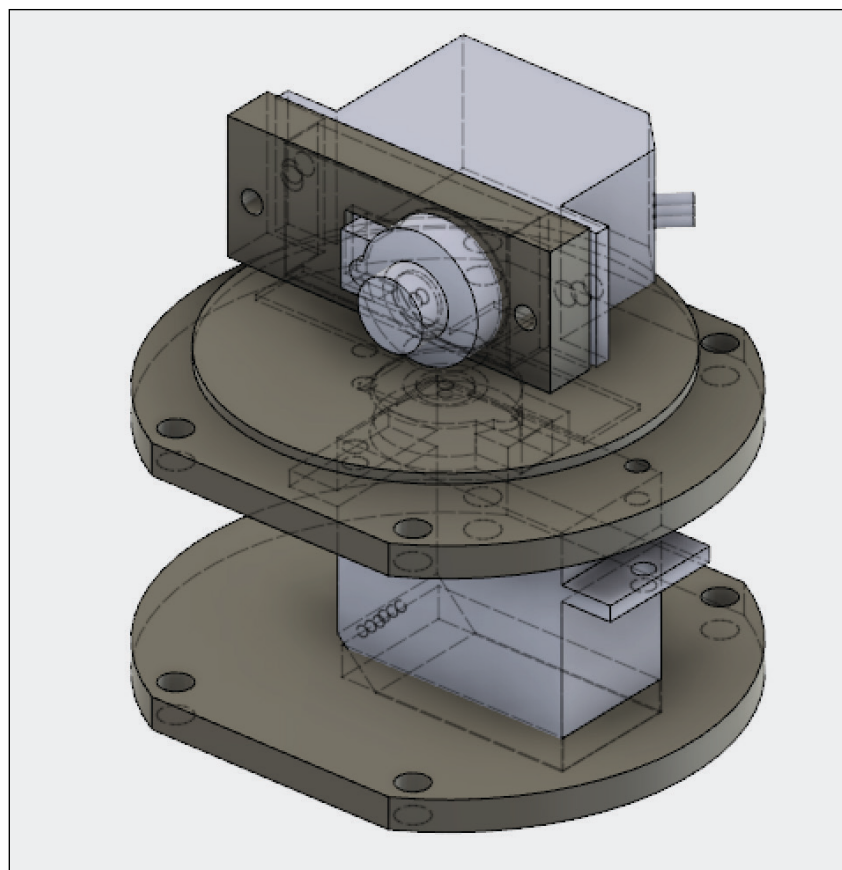


FIGURE 4

Shown here is the 3D CAD rendering of our base for the robotic arm. This joint needed to be 3D printed, because it required two servos to be attached in very close proximity—and hot glue alone wouldn't provide enough stability.

translate the IMU readings to motion in different axes. We gave constraints to each servo so that we don't surpass the angle of its physical limitations. In other words, if a human's elbow joint can't provide 360 degrees of rotation, our robotic arm shouldn't be able to do that either. Servo 1 rotates approximately 180 degrees, and Servo 2 and Servo 3 rotate 40 degrees and 90 degrees, respectively.

The most sophisticated component in our robotic arm was the base used to support Servo 1. It was 3D-printed. That's because we realized at the onset that we needed a solid, stable base to prevent movement from the other servos from destabilizing our system. The 3D rendering of the base is shown in **Figure 4**. Servo 1, which rotates the base, was fitted into a casing that was laser-cut to minimize jittering. Later, we attached the base casing to a plank of wood for greater stability, because the arms should have imbalance in their center of gravity. Then Servo 2 was screwed into a 3D-printed mold that fit right into Servo 1. The arms of the robots were extended by wooden coffee sticks. Finally, the holder or bowl of the ball was made with a plastic spoon. The "shoulder joint" was replicated by Servos 1 and 2. The elbow movement was replicated by Servo 3, which was attached at the end of the stick extended from Servo 2. The servos were connected to a board, which was connected to the PIC32 MCU used for the robotic arm station (**Figure 5**). The development board used here is Sean Carroll's PIC32 Large Development Board—a link with the details of this resource is provided on the *Circuit Cellar* article materials webpage.

COMPONENT BREAKDOWN

The IMUs used in this project were MPU6050 from TDK InvenSense, which uses I²C protocol for communication with the host device. The PIC32 has capability of two I²C channels. However, we only used one, because the MPU6050 has a bit address—0 or 1—which is used for talking to the specific IMU unit. We used a helper function, called `i2c_helper.h` from another project that also used I²C to communicate successfully with this sensor [3].

When we started experimenting with the data being collected, we faced a few problems with accuracy and drift. This is a common issue with IMUs, and was remedied by sensor fusion—using the information from both types of sensors in our IMU unit to correct the error. The Kalman Filter is the standard method to solve these problems, but it is computationally heavy and at times difficult to implement. A simpler algorithm is the complementary filter, which is easier to implement and is "often applied in systems of limited resources such as this project [4].

The complementary filter provided a simple way of getting accurate data and reducing drift. This was accomplished by combining the gyroscope and accelerometer data on each axis. This approach proved to be an efficient, computationally lightweight alternative to a

Kalman Filter for our system. We implemented the complementary filter for angle data for each joint. On the PIC32 attached to the sleeve, we extracted the most recent value for the accelerometer and gyroscope, scaled both of them and then processed them through our algorithm.

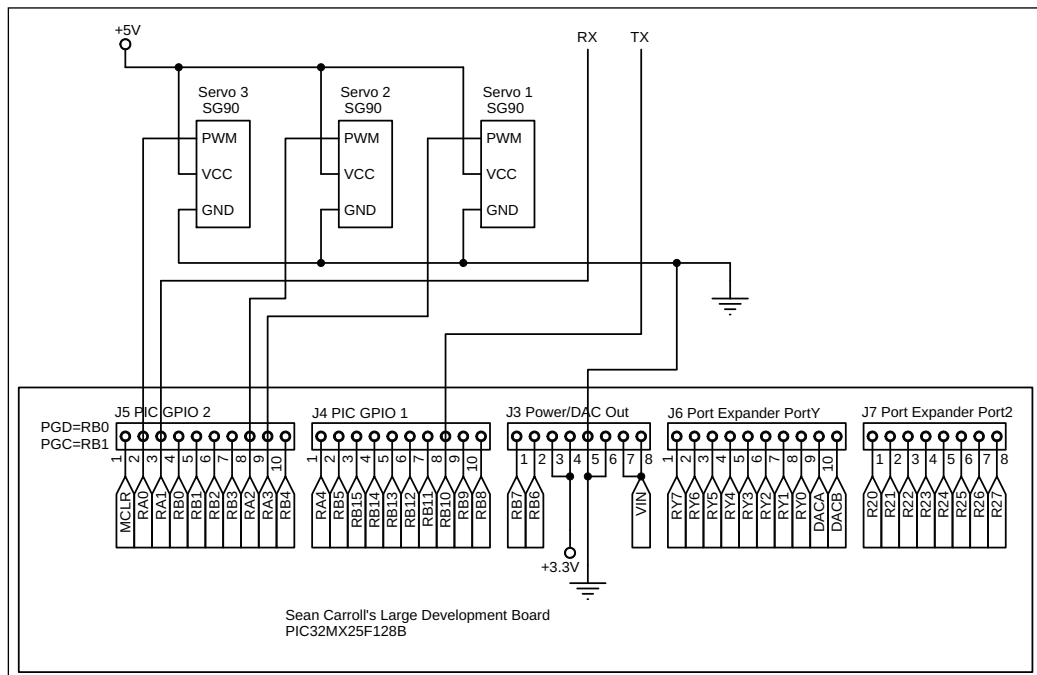


FIGURE 5

This is the schematic of the board and connections to the servos on the PIC32 used for the robotic arm. As shown in Figure 2, the XBee connections were ignored, because both PIC32s were connected physically by two wires to communicate via UART. These connections are shown at the top center of the diagram as RX and TX [2].

FEATURES

Equipment Cabinet for Industries that never stop

The R-Series from Optima-Stantron Proven, sturdy, configurable and ready to ship

Elma Electronic Inc., USA

With you at every stage | elma.com

```
// Elbow IMU
accTilty_elbow= -atan2f(xAccel_elbow,
zAccel_elbow)*180.0/M_PI;
tilty_elbow = (COMP_FILTER_G_COEF*(tilty_
elbow + yGyro_elbow*IMU_READ_PERIOD*0.001) +
COMP_FILTER_A_COEF*accTilty_elbow);
```

LISTING 1

This snippet shows the C code used to accurately calculate the elbow angle for the robotic arm. The variable *tiltY_elbow* was passed to the robotic arm.

Listing 1 shows a snippet of C code of the algorithm on our system to calculate the “elbow” angle from the sleeve. The past gyroscope data is integrated and then multiplied by the constant, `COMP_FILTER_G_COEF`, which is then added to the scaled accelerometer data. Multiplying this computed value by the other constant, `COMP_FILTER_A_COEF`, we get the final, filtered angle data. The values `COMP_FILTER_G_COEF` and `COMP_FILTER_A_COEF` (0.98 and 0.02 respectively) were used to weight the values of the accelerometer and gyroscope differently because of two things. First, the gyroscope drifts a lot when IMU is not detecting movement. Second, the accelerometer is easily disturbed, causing spikes in its readings—especially while facing movement. But it has data that could be useful to counteract the drift from the gyroscope. These can be adjusted, but 0.98 and 0.02 worked fairly well for us.

The pressure sensor was a variable resistor. To use it as a digital button, we built a simple voltage divider circuit and set a threshold. This meant that any time the output of the circuit exceeded it, we would set a flag. We used the ADC on the PIC32 to read voltage values to detect when to follow the arm and when to release the ball (or launch the catapult).

When we started debugging and testing our system, we needed a more reliable way of communication between PIC32 MCUs than RF because we also had to debug our RF communication. For this, we used UART between the PIC32s. This was convenient because our RF modules also used UART as a communication protocol to send data so, theoretically, we could see them as a UART bridge.

The servo motor controls were all written on the PIC32 MCU that was hooked up to the

robotic arm. Its purpose was to extract all the data that were sent from the PIC32 MCU on the sleeve and turn them into PWM signals. Four variables were extracted from the PIC32 on the sleeve. Three represented the tilt of elbow on Z axis, on Y axis, and tilt of wrist on Z axis. And the fourth was a flag indicating whether or not the pressure sensor was pressed.

RESULTS AND USABILITY

We performed sets of testing to deduce numerical specifications for our system. Drift tests were done for each servo experimentally, to observe drift in angles. For each test we reset the system and performed 20 cycles of the maximum range of motion allowed for each degree of freedom. On Servo 1—which was getting data from the IMU on the elbow—we observed approximately an 8-degree difference when returning to the original position. Servo 2 used data from the same IMU, and we found no measurable drift after our test. This was reasonable given that the range of motion for this servo was only 40 degrees. On Servo 3—which used data from the IMU on the wrist—we also saw no significant drift after the 20 cycles and returning to the original position.

While testing the Servo 2 and Servo 3 rotation, we noticed that Servo 1 was also rotating slightly, which should have not been the case. This may have been due to IMU’s position on the elbow, which wasn’t securely fixed at one position. The arm movement made the sleeve elastically extend and contract, and in this process the IMU might effectively be moved around. This might also have caused movement to the base rotation and stacking up drift angles.

To determine how much weight our delightfully crafted robotic arm could handle, we performed a series of tests with increasingly heavier materials. Although the arm’s intended use was to throw lightweight objects such as ping pong balls, we found that any object lighter than 15 g also could be thrown in an acceptable way.

Finally, we tested the range of throw to specify the shortest and longest distance the throw could cover. We found that by adjusting the initial position of the throw, we could cover a range of 6" to 13". This might not be a lot—especially if you’re trying to play a real game of beer pong against a real person—but it could certainly bring some external entertainment to the game.

As for speed of execution, as noted previously, wired communication was more effective. Combining that with the use of interrupts with UART, we were able to get our robotic arm to mimic the user’s arm with no noticeable delay. This is mainly because we

Additional materials from the authors are available at:

www.circuitcellar.com/article-materials

References [1] through [4] as marked in the article can be found there

RESOURCES


Digi International | www.digi.com

Microchip Technology | www.microchip.com

TDK InvenSense | www.invensense.com

don't expect the users to make any sudden movements while aiming. We noticed that quick, sudden movements made the robotic arm struggle to catch up, and in some cases, it reacted in strange ways that disappeared when the user started aiming more slowly.

Anyone with an arm and a finger can use our controller, as long as the IMUs are adjusted to match the location of the elbow and the wrist. Also, the robotic arm is quite fragile because it's made of birchwood sticks and cheap generic servos that seem to jitter slightly in motion. The robustness of the

robotic arm can be improved by replacing the birchwood sticks with 3D-printed parts and by using more reliable servos. The arm controller on the sleeve can also be made more robust by strapping components more tightly on the user's arm. Lastly, the user must acknowledge and consider the slight drifting on the base rotation of the robot and the maximum load the system can take to throw. A video of the final demo can be found by scanning the QR code at the end of this article (**Figure 6**) or you can find the video on *Circuit Cellar's* article materials webpage. 

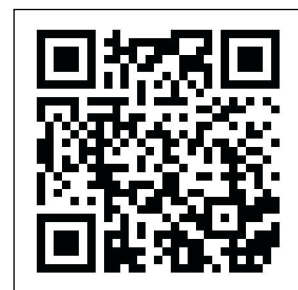


FIGURE 6

A video with a demo of the final version of the project can be found by scanning this QR code.

About the Authors

Daniel Fayad graduated Cornell in May 2018 with a degree in Electrical and Computer Engineering. He now is a Software Development Engineer on Amazon's Alexa team. He is interested in Embedded Systems, Speech Recognition, Computer Vision and he isn't particularly good at beer pong.

Harrison Hyundong Chang graduated from Cornell University in 2017 with bachelor's and master's degrees in Mechanical and Aerospace Engineering. He worked as an Antenna Engineer for a year at Samsung Electronics, mobile communications business, and is now working as a Hardware Engineer for an Internal Corporate Venture called C-Lab at Samsung Electronics.

Justin Choi obtained a bachelor's degree in Mechanical Engineering from Cornell in 2017, followed by a master's degree in Aerospace Engineering from Cornell in 2018. He now works at Northrop Grumman Innovation Systems as a Guidance Navigation and Control Engineer, working on development of flight computer algorithms for commercial and science satellites.

AdaCore



**25 YEARS OF HELPING OUR CUSTOMERS BUILD
SAFE, SECURE AND RELIABLE SOFTWARE**

Digital Signage Technologies Gain Momentum

System Solutions



Digital signage ranks among the most dynamic areas of today's embedded computing space. Vendors involved in this technology continue to roll out new solutions for developing powerful digital signage implementations.

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

Digital signage is one of those technologies that seemed to breeze into our modern society so quickly and smoothly that it's hard to image life without it. Today's technologies provide users with the ability to easily update information on large, high-resolution displays in real-time and in rugged, outdoor environments. And the ability to rotate ads even on billboard-sized displays has multiplied revenue streams for stakeholders using digital signage systems.

At the heart of today's landscape of modern digital signage are a variety of digital signage players that support advanced graphics and multiple streams of connectivity. Also in the mix are general-purpose box-level embedded computing systems that provide solutions for signage applications. Obviously displays make up part of the ecosystem too, but this article focuses strictly on the embedded computing side of digital signage.

WATERPROOF DESIGN

In March, Ibase Technology launched its latest SW-101-N waterproof digital signage player designed for both indoor and harsh

outdoor environments. This rugged fanless signage player is integrated with a 1.91 GHz Intel Atom Processor E3845 Quad-Core Processor and Intel HD graphics (Gen 7-LP) 4EU (**Figure 1**). The SW-101-N is built to withstand dust, water and extreme temperatures. This ensures the system's stable operation and reliability in harsh industrial environments.

The SW-101-N meets IP68 standards, allowing it to handle submersion in water for up to 30 minutes at a depth of 1.5 meters. The black-color waterproof enclosure uses a C3 HDMI connector and M12 I/O interface connectors for two USB 2.0, one Gbit LAN, one RS-232, DC power input and digital I/O. Two antenna N-jack type connectors have waterproof designs as well. Aside from being fanless, the unit has a wide operating temperature range of -40°C to 75°C.

The SW-101-N supports Ibase's iControl and Observer technologies for intelligent control and remote monitoring functions that feature auto power on/off scheduling, power resume, system temperature/voltage remote monitoring and low temperature boot protection. The standard model has 4 GB of DDR3L-1333 system memory, 64 GB

mSATA storage, and 12 V DC-in support. Additional features include a watchdog timer, wall mounting and Mini PCIe expansion for optional wireless modules.

TINY SIGNAGE PLAYER

A powerful set of digital signage functionality can be squeezed into a very small form factor these days. In an example along those lines, in September Advantech introduced its USM-110, an ultra-compact digital signage player. This fanless system provides support for Android 6.0 and Advantech's own WISE-PaaS/SignageCMS digital signage management software. The compact (156 mm x 110 mm x 27 mm) device follows earlier Advantech signage computers such as the slim-height, Intel Skylake based DS-081.

The USM-110, which is also available in a less feature rich USM-110 Delight model, ships with 2 GB DDR3L-1333, as well as a microSD slot. It has 16 GB of eMMC on the standard version and 8 GB on the Delight. There's also a GbE port and an M.2 slot with support for an optional Wi-Fi module with antenna kit.

The USM-110 has two HDMI ports, both with locking ports: an HDMI 2.0 port with H.265-encoded, native 4K at 60 Hz (3840 x 2160) and a 1.4 port with 1080p resolution. The system enables dual simultaneous



FIGURE 1

The SW-101-N is a waterproof, fanless signage player that is integrated with a 1.91 GHz Intel Atom Processor E3845 Quad-Core Processor and Intel HD graphics (Gen 7-LP) 4EU. The SW-101-N is built to withstand dust, water and extreme temperatures in mind.

HD displays. The Delight version lacks the 4K-ready HDMI port, as well as the standard model's mini-PCIe slot, which is available with an optional 4G module with antenna kit. The Delight is also missing the standard version's RS232/485/422 port, and it has only one USB 2.0 host port instead of four. Otherwise, the two models are the same, with a micro-USB OTG port, audio jack, reset, dual LEDs and a 12V/3A DC input. The 0.43 kg system has a 0 to 40°C range, and offers VESA, wall, desktop, pole, magnet and DIN-rail mounting (**Figure 2**).

Advantech's WISE-PaaS/SignageCMS digital signage management software—

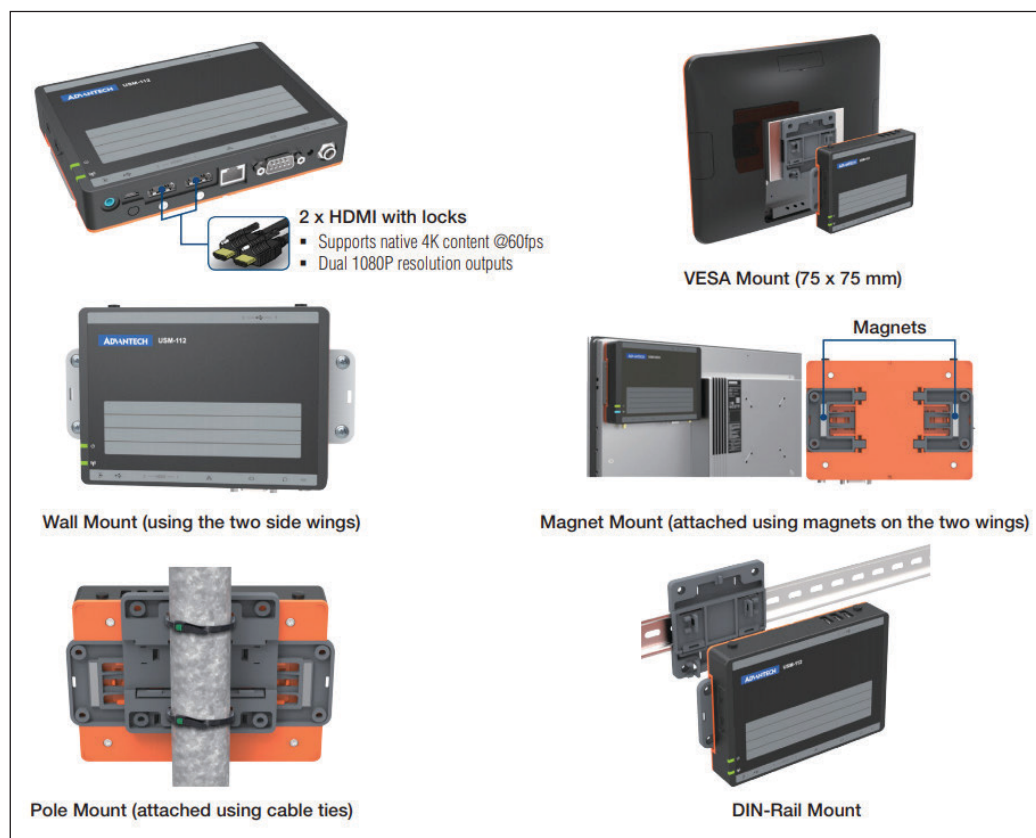


Figure 2

The USM-110 is a digital signage player that supports Android 6.0 and Advantech's WISE-PaaS/SignageCMS digital signage management software. The compact unit measures 156 mm x 110 mm x 27 mm and features VESA, wall, desktop, pole, magnet and DIN-rail mounting options as shown here.

SPECIAL FEATURE

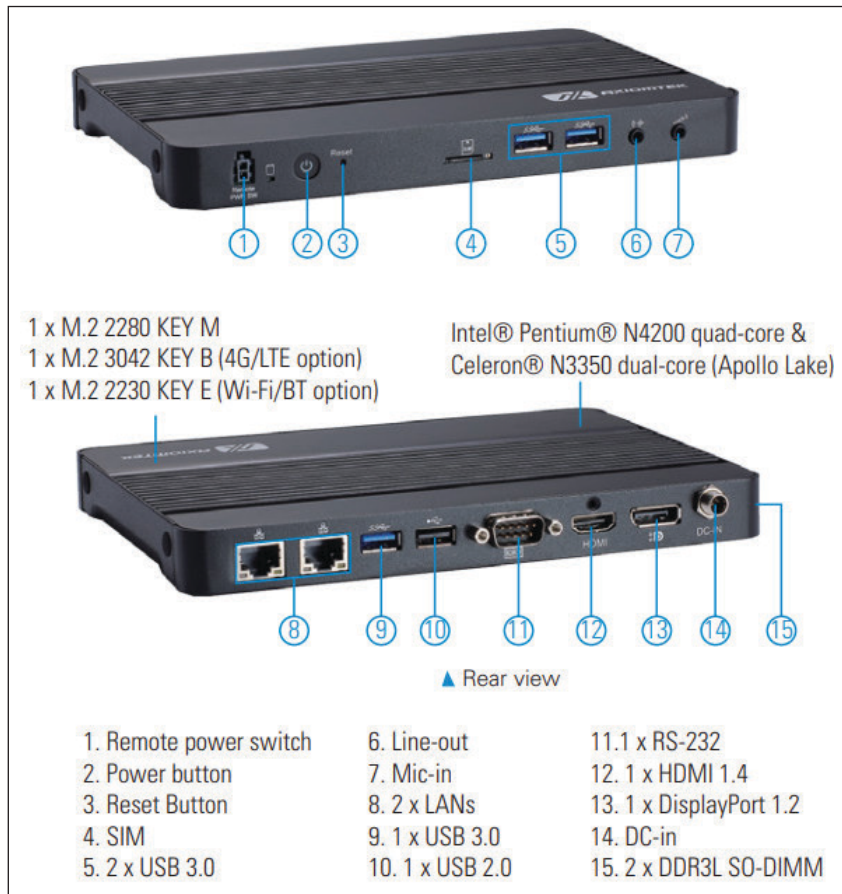


Figure 3

DSP300-318 is an Intel Apollo Lake based digital signage player with two DDR3L-1600 SO-DIMMs providing up to 8 GB of system memory. There's an M.2 E-Key 2230 for Wi-Fi and Bluetooth and an M.2 B-Key 3042 for 4G LTE. A SIM card slot and 4x antenna mounts are also available.

also referred to as UShop+ SignageCMS—supports remote, real-time management. It allows users to layout, schedule and dispatch signage contents to the player over the Internet, enabling remote delivery of media and media content switching via interactive APIs. A WISE Agent framework for data acquisition supports RESTful API web services for accessing and controlling applications.

CABLE-FREE DESIGN

Like many of today's embedded applications, digital signage has entered the wireless era. Along just those lines, in February Axiomtek launched the DSP300-318, an Intel Apollo Lake based digital signage player promoted for its ultra-slim, 200 mm x 137.8 mm x 20 mm dimensions. The 4K-ready system is designed for space-constrained digital menu boards, self-ordering systems, retail applications, queuing systems, interactive kiosks and video walls.

The system runs Ubuntu or Debian Linux or Windows 10 IoT on Intel's dual-core, 1.1 GHz Celeron N3350 or quad-core, 2.5 GHz Pentium N4200. Two DDR3L-1600 SO-DIMMs provide up to 8 GB of system memory. And there's an option for 64 GB eMMC 5.0. The

DSP300-318 stands out with its triple M.2 slot design. In addition to an M.2 M-Key 2280 for storage, there's an M.2 E-Key 2230 for Wi-Fi and Bluetooth and an M.2 B-Key 3042 for 4G LTE. A SIM card slot and 4x antenna mounts are also available (**Figure 3**).

The DSP300-318's 4K-ready HDMI 1.4 and DisplayPort 1.2 ports support dual simultaneous displays. Other features include 2x GbE ports, 3x USB 3.0 ports and single USB 2.0 and RS-232 ports. Dual audio jacks are also available. The DSP300-318 has a 12 VDC terminal screw input, as well as power, reset, and remote switches. There's also a watchdog timer and a Lithium 3V/220mA-hour battery. The fanless system supports 0 to 50°C temperatures and offers humidity resistance and 3 Grms vibration resistance with M.2 storage (5 to 500 Hz, X, Y, Z).

PLAYERS WITH OPS SUPPORT

In 2010, Intel launched the Open Pluggable Specification (OPS) to standardize the system architecture between displays and media players. According to Intel, OPS allows for more cost-effective design, deployment, and management of digital signage and other display solutions that support advanced functionality and emerging use cases, including interactivity and anonymous audience analytics. OPS began appearing in signage systems such as the Axiomtek OPS860 back in 2011. The spec standardizes mounting and power requirements and connects to OPS-compatible displays via an 80-pin JAE Electronics TX24/TX25 blind mate plug and receptacle connector system.

In June 2018, Ibase launched its IOPS-602 signage player that runs Windows 10 or Ubuntu Linux on Intel's 6th or 7th Gen. Core QC/DC processors, with a default to dual-core, 7th Gen "Kaby Lake" U-series processors with 15 W TDPs. The standard SKU is a Core i7-7600U (2.8 GHz/3.9 GHz) with 8 GB RAM and 128 GB of M.2 storage.

The 200 mm x 119 mm x 30 mm IOPS-602 uses an OPS standard 12 V to 19 V DC input and OPS mounting bracket. The JAE connector is mounted on the back of the system. An optional expansion dock with 150 W adapter is available for using the systems with non-OPS displays. Up to 32 GB of DDR4-2133 DRAM can be loaded via dual slots, and there's an M.2 M-Key slot for 2280 SSD cards. An M.2 E-Key slot is available for 2230-based Wi-Fi/Bluetooth cards.

The IOPS-602 also provides 4x USB 3.0,

HDMI 1.4b and Gbit Ethernet ports, as well as an RS232 serial connection provided via an RJ45 port. You also get dual audio jacks, LEDs, a watchdog and iAMT compliance for remote management. The system supports 0°C to 45°C temperatures and resists vibrations to the tune of 5 Grms, 5 to 500 Hz random operation with an SSD.

MOVING ON TO OPS+

Intel developed a follow-on spec called OPS+ that builds on the benefits and powerful functionality of the OPS by enabling a broader range of Intel processors to include the Intel Xeon processor family, a range of Intel desktop processors and Intel FPGAs. OPS+ can also add functionality based on specific industry needs such as supporting simultaneous display and broadcast usages, support for 8K resolution displays and the ability to drive three individual 4K resolution display outputs.

According to Intel, OPS+ defines a 180 mm x 119 mm x 30 mm, fully enclosed digital signage systems with enhanced thermal design supports broader range of Intel processors. The enhanced spec is optimized for interactive white boards (IWBs), commercial digital signage, kiosks, visual data devices, video walls and so on. With OPS+, you can customize a protocol and simultaneously support advanced use cases including real-time analytics and video capture performed on the display itself. The spec also features a second high-speed connector and is backward compatible with previous OPS specifications.

In December Axiomtek released the first OPS+-compliant digital signage player, the OPS700-520. The system is powered by the LGA1151 socket 8th generation Intel Core i7/i5/i3 and Celeron processors (codename: Coffee Lake S) with the Intel Q370 chipset. The player supports Intel Active Management Technology (Intel AMT) 11.0 as well as Intel Unite solution for content sharing and collaboration. It comes with two 260-pin DDR4-2400 SO-DIMM sockets that can provide system memory of up to 32 GB (**Figure 4**).

The OPS700-520 is compatible with Intel Unite, which allows users to connect and interact with meeting content in real time, thus enhancing seamless meeting experiences and convenience. It also comes with Intel AMT 11.0. Software issues can be repaired wirelessly while failed hardware components can be identified beforehand, thereby lowering maintenance costs and improving efficiency. The signage module is

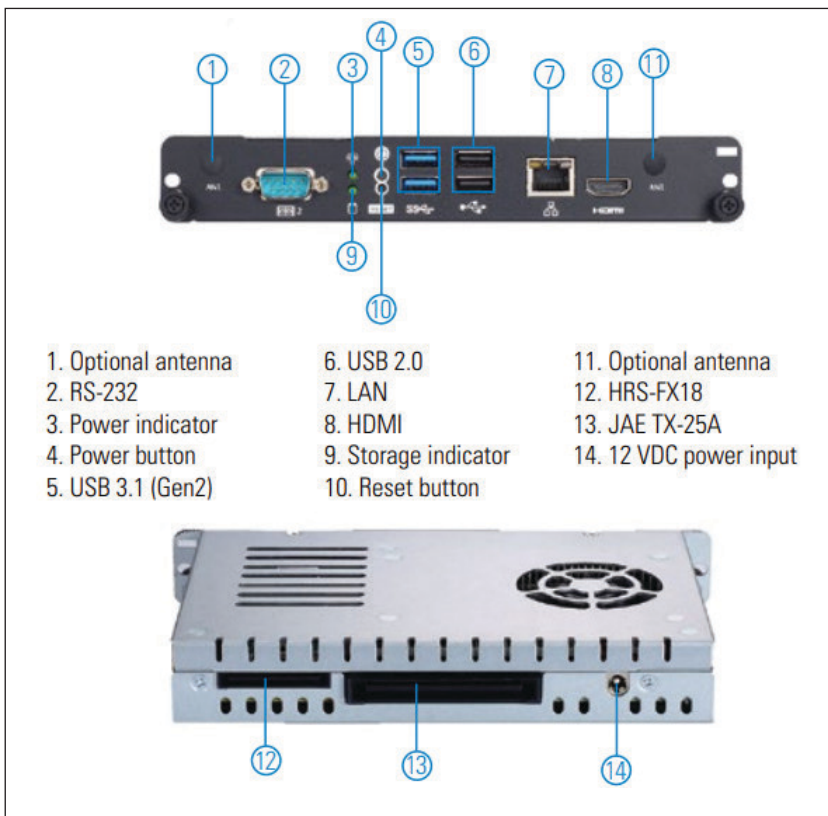


Figure 4 The first OPS+-compliant digital signage player, the OPS700-520 is powered by the LGA1151 socket 8th generation Intel Core i7/i5/i3 and Celeron processors. The player supports Intel Active Management Technology (Intel AMT) 11.0 as well as Intel Unite solution for content sharing and collaboration.

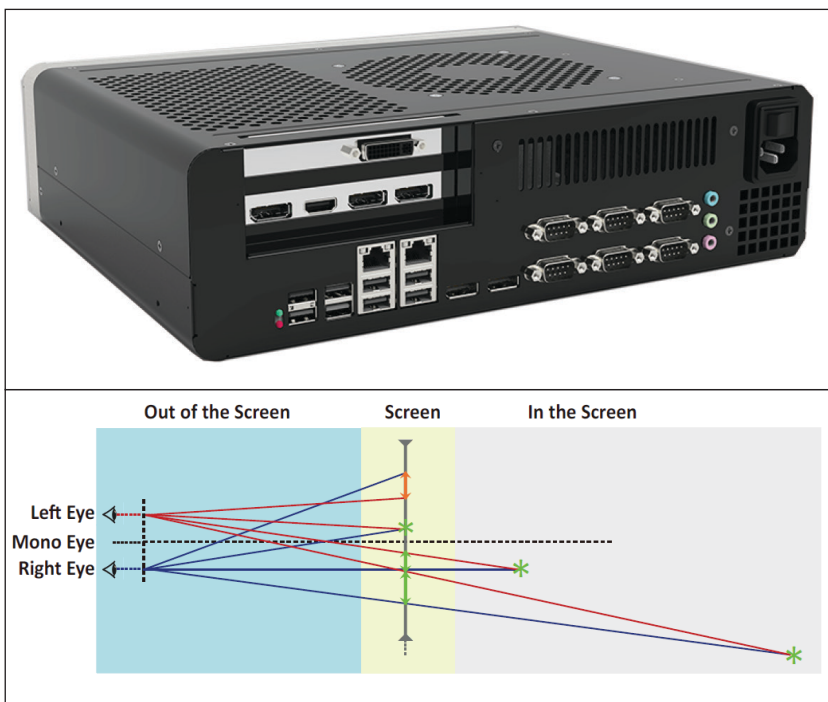


Figure 5 3D Bare-Eye Content Development Kit and Signage Solution was designed to enable developers of casino slot machine games and digital signage displays to provide 3D content that can be viewed without special glasses.

SPECIAL FEATURE



Figure 6

The ML100G-31 embedded PC system is built around an Intel Dawson Canyon NUC board and employs the company's Hardshell Fanless Technology to ensure thermal performance.

suitable for multi-display solutions such as IWBs in meeting rooms, commercial digital signage, video walls and more.

The digital signage player can be easily connected to an OPS-plus compliant display via two high-speed transmission connector interfaces: JAE TX25A and HRS-FX18. The JAE plug connector interface supports one DisplayPort (4K at 60 Hz), one HDMI 2.0 (4K at 60 Hz), one USB 3.0, two USB 2.0, one audio and UART signals. The HRS plug connector interface supports one DisplayPort (4K at 60 Hz) and one PCI Express x4. These two connector interfaces enhance multimedia performance to meet various requirements. The OPS700-520 also has one PCIe or SATA interface for storage, one M.2 Key E for Wi-Fi modules and one M.2 Key M NVMe SSD slot.

The OPS700-520 maintains the small form factor with dimensions of just 200 mm x 119 mm x 30 mm. It comes with rich I/O connectors including two USB 3.1 Gen2, two USB 2.0, one RS-232 (COM 2), one Gbit LAN with Intel i219-LM Ethernet controller and one HDMI. The unit supports Windows 10 64-bit and Linux operation systems. Also, it supports the TPM 2.0 which can provide security and privacy benefits.

3D DIGITAL SIGNAGE

A unique twist on tradition digital signage in the emergence of 3D capability. Feeding that need, in October last year EFCO introduced a development and signage solution for creating advanced 3D slot machine games. The

company's 3D Bare-Eye Content Development Kit and Signage Solution was designed to enable developers of casino slot machine games and digital signage displays to provide 3D content that can be viewed without special glasses (**Figure 5**).

3D Bare-Eye is based on the Unity software environment, which, according to EFCO, is the defacto standard development toolset among game developers. When used for casino games, instead of simply displaying images of coins on the screen, the coins now appear to be falling out of the slot machine toward the player. But the technology can also be used for any digital signage or progressive display application, says EFCO.

The 3D Bare-Eye Solution is made up of a development kit and a broadcast kit. The content development kit is based on Unity. Because Unity is the most common gaming development environment, it's easy to adopt. The kit also comes with a monitor, computer system and a proprietary interface card that connects the development system to the playback system. Features of the 3D Content Development kit include: Intel Core i5-6500, 4C/4T with boost to 3.6 GHz, NVIDIA GTX1050Ti (4 GB GDDR5) or GTX1070Ti, 2.5" SATA SSD 256 GB, an average 190 W power consumption and 3840 x 2160 display support.

The broadcast kit comes with a ready-to-use 55" and 65" 3D digital signage 4K display with playback system. A 3D film on the monitor provides the third dimension to viewers. Features of the kit include Intel Pentium CPU, NVIDIA graphics GTX1050Ti (4GB GDDR5), 2.5" SATA SSD 64 GB storage, power input of AC 100 V to 240 V, 50 Hz to 60 Hz and power consumption averaging 150 W.

FANLESS SOLUTION

While dedicated, purpose-built solutions—like the ones discussed so far in this article—are one approach to digital signage applications, another angle is to employ box-level general purpose embedded computers to serve the player functionality. This approach makes sense especially when extreme environmental conditions are an issue. An example along these lines is Logic Supply's ML100G-31 embedded PC system introduced last August. This system is built around an Intel Dawson Canyon NUC board and employs the company's Hardshell Fanless Technology to ensure thermal performance. Logic Supply says it's the smallest fanless and ventless NUC to feature an 8th generation (Kaby Lake) Intel

RESOURCES

AAEON | www.aaeon.com

Advantech | www.advantech.com

Axiomtek | us.axiomtek.com

EFCO | www.efcotec.com

Ibase Technology | www.ibase.com.tw

Intel | www.intel.com

Logic Supply | www.logicsupply.com

Core i7 processor (**Figure 6**).

The ML100G-31 provides a fully solid state, passively cooled computing solution, designed for reliability in demanding environments and measures just 142 mm x 62 mm x 107mm. Logic Supply engineers, with support from Intel's thermal design lab, created a proprietary heatsink for the NUC717DNBE motherboard and Quad-Core i7-8650U Kaby Lake CPU. They also collaborated with Intel to identify a way to ensure that the ML100G-31 features the 5-year lifecycle that will allow their industrial computing clients to standardize on the platform.

The ML100 is able to cool the processor and other internal components by employing Logic Supply's proven Hardshell Fanless Technology. Through the use of unique exterior fins and specially machined heatsink design, the system is able to maintain an optimal operating temperature without the need for a cooling fan. Removing the fan from the system improves overall reliability. Unlike fanned solutions that are vulnerable to airborne contaminants, this fanless design is able to operate in challenging computing environments across a range of industries including manufacturing and automation, industrial digital signage and others.

The system can be configured with up to 32 GB of memory and 1 TB of M.2 storage. Connectivity includes four USB 3.0 ports, two HDMI ports supporting dual 4K output, Gbit LAN and an optional COM port for legacy equipment connectivity. Operating system options include both Windows and Linux Ubuntu.

SYSTEM BASED ON MINI-ITX

In another example of a general-purpose system that's suited for digital signage, AAEON in September released the ACS-1U01 Series, a range of turnkey solutions that embed three of its bestselling SBCs. By enclosing the boards inside a tough 1U chassis, the unit provides a ready-to-go system for use in a variety of applications including digital signage as well as industrial automation, POS, medical equipment and transportation.

The three models—the ACS-1U01-BT4 (**Figure 7**), ACS-1U01-H110B, and ACS-1U01-H81B—feature a tough, 44.45 mm-high chassis with a wall mount kit and 2.5" HDD tray. The low-profile, low-power-consumption systems have full Windows and Linux support, they can be expanded via full- and half-size Mini-Card slots and heatsinks give them operating temperature ranges of 0°C to 50°C.



The ACS-1U01-BT4 houses AAEON's EMB-BT4 motherboard, which can be fitted with either an Intel Atom J1900 or N2807 processor. The J1900 can be used with a pair of DDR3L SODIMM sockets for up to 8 GB dual-channel memory, while the N2807 can be used with a single DDR3L SODIMM socket. The board's extensive I/O interface provides the system with a GbE LAN port, dual independent HDMI and VGA displays, a USB3.0 port, up to seven USB 2.0 and up to six COM ports.

The ACS-1U01-H110B contains AAEON's EMB-H110B, which is built to accommodate up to 65 W 6th/7th Generation Intel Core i Series socket-type processors and supports up to 32 GB dual-channel memory via a pair of DDR4 SODIMM sockets. Dual independent display support is possible through two HDMI ports, or the option of DP connections. The system also features a GbE LAN port, four USB 3.0 ports, four USB 2.0 ports and a COM port.

The ACS-1U01-H81B is built around AAEON's EMB-H81B, which is designed for 4th Generation Intel Core i Series socket-type processors with TDPs of up to 65 W. Two SODIMM sockets allow for up to 16 GB dual-channel DDR3 memory, and HDMI, DisplayPort and optional VGA ports enable dual independent display. The system has two GbE LAN ports, two USB3.0 ports and six USB 2.0 ports.


There's no doubt that digital signage is an application that puts high demands on a variety of technology segments—from graphics processing to connectivity to form factor design. To keep pace with demands, makers of digital signage players and embedded PCs continue to innovate by adding more capabilities while also shrinking size, weight and power. 

Figure 7

The ACS-1U01-BT4 houses AAEON's EMB-BT4 motherboard, which can be fitted with either an Intel Atom J1900 or N2807 processor. The J1900 can be used with a pair of DDR3L SODIMM sockets for up to 8 GB dual-channel memory, while the N2807 can be used with a single DDR3L SODIMM socket.

its capabilities. Developers can use C-RUN in C-SPY simulator as well as in their actual target hardware. IAR Systems provides a size-limited version of C-RUN that is activated for evaluation when you download IAR Embedded Workbench for Arm V7.20 (and later versions) or IAR Embedded Workbench for RX V3.10.

ADA STATIC ANALYSIS

With its main focus on Ada language tools, AdaCore’s static analysis tool suite is called CodePeer. CodePeer is an Ada source code analyzer that detects run-time and logic errors. It assesses potential bugs before program execution, serving as an automated peer reviewer, helping to find errors easily at any stage of the development life-cycle (Figure 2).

CodePeer is a stand-alone tool that runs on Windows and Linux platforms. It may be used with any standard Ada compiler or fully integrated into the GNAT Pro development environment. It can detect several of the “Top 25 Most Dangerous Software Errors” in the Common Weakness Enumeration. CodePeer supports all versions of Ada (83, 95, 2005, 2012). CodePeer has been qualified as a Verification Tool under the DO-178B and EN 50128 software standards.

In February, AdaCore released Version 19.1 of its flagship products including CodePeer as well as its GNAT Pro, CodePeer, SPARK Pro and QGen products. The enhancements in CodePeer 19.1 are focused on user/usability improvements. These include new entry level (“level 0”) with fast analysis and minimal false positives. A simple “getting started quickly” mode is provided for new users. Other new features include a security report output, integration of AdaCore’s GNATcheck tool and a major documentation update—including examples of typical workflows.

INTEGRATED WITH COMPILER

As one of the long time veterans in the embedded software industry, Green Hills Software provides its MULTI IDE that includes a rich set of debugging and analysis tools. Among these are its DoubleCheck integrated static analysis tool. Green Hills emphasizes the importance of this tool as being an integrated tool. In other words, DoubleCheck is built into the Green Hills C/C++ compiler—unlike other source code analyzers that run as separate tools.

A typical compiler issues warnings and errors for some basic potential code problems, such as violations of the language standard or use of implementation-defined constructs. In contrast, DoubleCheck performs a full program analysis, finding bugs caused by complex interactions between pieces of code that may not even be in the same source file.

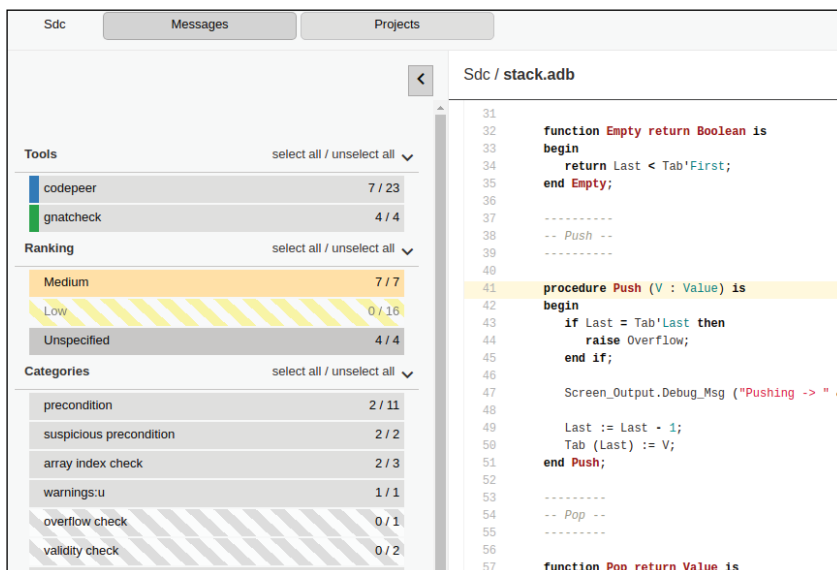


FIGURE 2 CodePeer is an Ada source code analyzer that detects run-time and logic errors. It assesses potential bugs before program execution, serving as an automated peer reviewer.

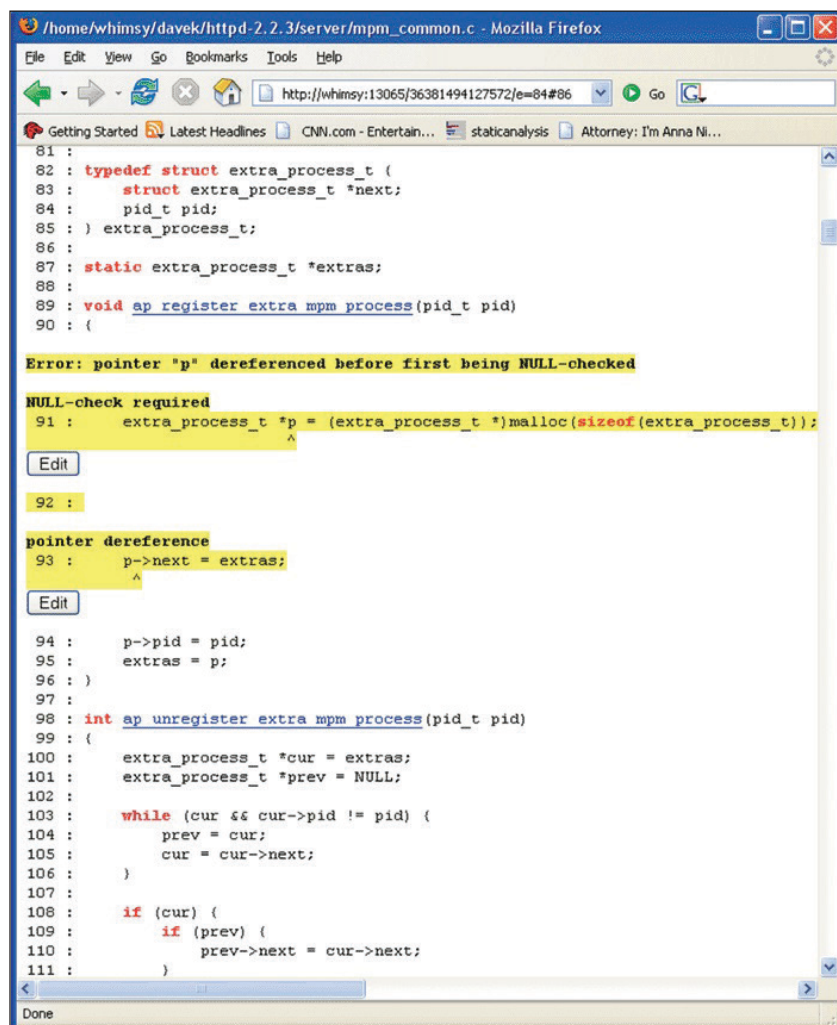


FIGURE 3 DoubleCheck determines potential execution paths through code. As shown there, errors found by DoubleCheck are displayed inline with the surrounding code, making them easy to understand.

DoubleCheck determines potential execution paths through code, including paths into and across subroutine calls, and how the values of program objects—such as standalone variables or fields within aggregates—could change across these paths (**Figure 3**).

Examples of the types of flaws DoubleCheck looks for are potential NULL pointer dereferences, buffer overflow, potential writes to read-only memory, resource leaks and others. The analyzer understands the behavior of many standard runtime library functions. For example, it knows that subroutines like `free` should be passed

pointers to memory allocated by subroutines like `malloc`. The analyzer uses this information to detect errors in code that calls or uses the result of a call to these functions.

Software development organizations often employ an internal coding standard which governs programming practices to help ensure quality, maintainability and reliability. DoubleCheck can automate the enforcement of these coding standards. For example, DoubleCheck has a Green Hills Mode that adds a range of sensible quality controls to its bug-finding mission, including several MISRA compliance checks, enforcement of optional but important language standards and more.

Metric computations and enforcement of other coding rules do not incur significant overhead since DoubleCheck is already traversing the code tree to find bugs. DoubleCheck can be configured to generate a build error that highlights problem code to keep developers from accidentally submitting software that violates the coding rules. Using DoubleCheck as an automated software quality control saves the time and frustration typically associated with peer reviews.

PRE-DEBUG ANALYSIS

For its part, Segger Microcontroller also provides static analysis as part of its IDE, Embedded Studio (**Figure 4**). Embedded Studio is a complete development environment for any Arm based processor, from legacy Arm7, Arm9 and Arm11 devices to Cortex-A, R and M. It comes with a system library that is optimized for embedded systems and GCC and LLVM/Clang compilers.

Embedded Studio offers various features and windows that provide you with enough information to analyze your application even before debugging. The Memory Usage Window goes into detail to show you where the sections—code and data—are placed. The Code Outline Window presents a clear structured outline of your source, which eases navigation through your code. The Source Navigator feature provides fast access to all your functions typedefs and variables with a single click. The Symbol Browser provides more insight into the compiled application. You can see how much memory is used by each symbol and where it will end up in your target. The Stack Usage Window does a static stack analysis of your application and shows the stack use of functions and call paths.

The Code Analyzer in Embedded Studio goes beyond the typical compiler warnings of an IDE. A compiler will usually generate warnings for anything that might break your application, such as uninitialized variables. To find further issues which have no immediate effect but might affect performance—and to

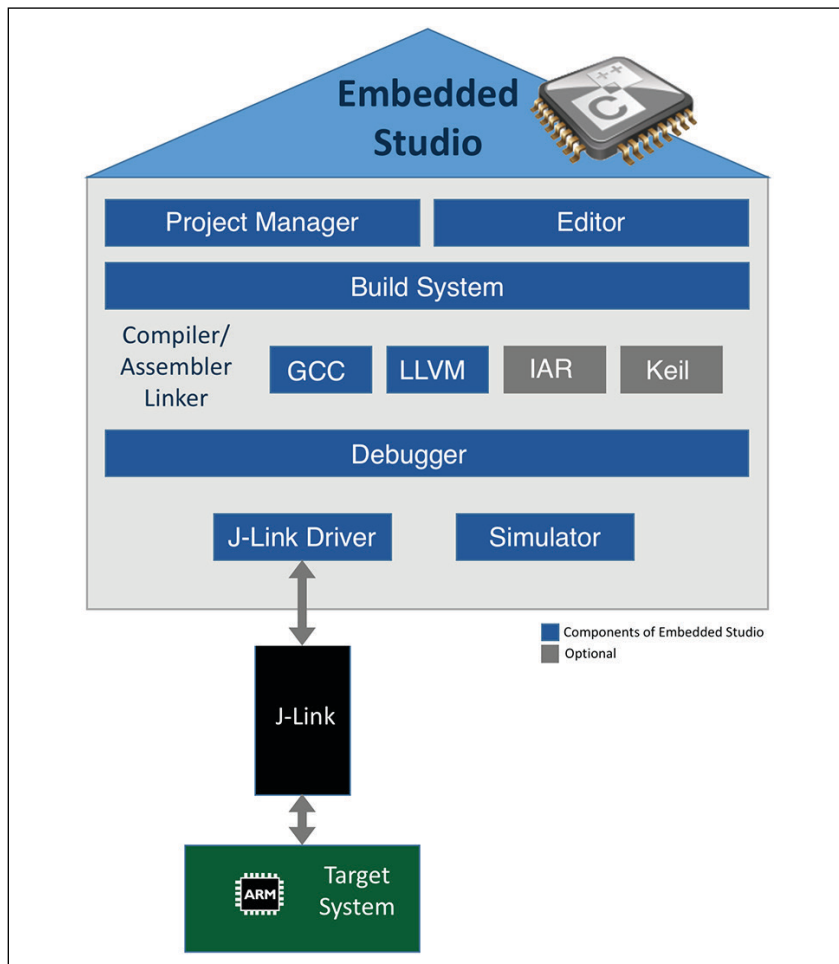


FIGURE 4

A common strategy is to integrate code analysis as part of an IDE. The Embedded Studio IDE does this with a Code Analyzer feature that goes beyond the typical compiler warnings of an IDE. To increase your code quality, you can run the IDE's Code Analyzer tool on your sources and findings will be logged and displayed.

RESOURCES

AdaCore | www.adacore.com

GrammaTech | www.grammatech.com

Green Hills Software | www.ghs.com

IAR Systems | www.iar.com

LDRA | www.ldra.com

Segger Microcontroller | www.segger.com

increase your code quality—you can run the Code Analyzer analysis on your sources. All findings will be shown in the log to easily navigate to them.

GOING DEEPER FOR IoT

Unlike many of the other vendors covered in this article, GrammaTech is not an IDE vendor. Instead, it specializes in code analysis with an emphasis on deep code analysis. In February, the company announced the latest release of its CodeSonar tool, version 5.1, with a focus on the Internet of Things (IoT). The new version of CodeSonar is designed to provide IoT developers the capability to support their multitude of languages and deliver safer and more secure software products faster.

With CodeSonar, developers can use a single user interface to find, assess and correct security vulnerabilities in different programs using multiple programming languages. CodeSonar 5.1 is tightly integrated with the Julia engine from Juliasoft, which provides high recall, high precision detection of security vulnerabilities in Java and C#. For developers of IoT systems, this is critical because IoT devices and enterprise services are built using many different programming languages. While C# or Java are typically the languages used on the user-interface or enterprise side, the embedded device itself is built using C/C++, with Python in the mix for scripting.

CodeSonar's Qualification Kit is available as an add-on for software developers that have requirements to support functional safety standards such as IEC 61508, DO178B/C or ISO 26262. The Qualification Kit enables developers to qualify CodeSonar in their environment as a preparatory step in the safety certification process. CodeSonar now supports the import and export of results in SARIF (Static Analysis Results Interchange Format).

A new API Anomaly detection module is now included CodeSonar, which uses statistical machine learning to distill checkers from open source bodies of code. This module reports reliability and security problems due to bad use of 3rd party APIs such as the GNU C Library, OpenSSL, Qt, Glib, GTK, libXML and others. This module has already been used to report problems in the Git version control system, the elinks browser, the Query Object Framework, Gnome and other projects.

AVOIDING LANGUAGE PITFALLS

Some programming languages, particularly C and C++, include features that are prone to causing problems. **Figure 5** shows output from LDRA's static analysis tools, as it relates to adherence to a MISRA language subset. MISRA—like other coding

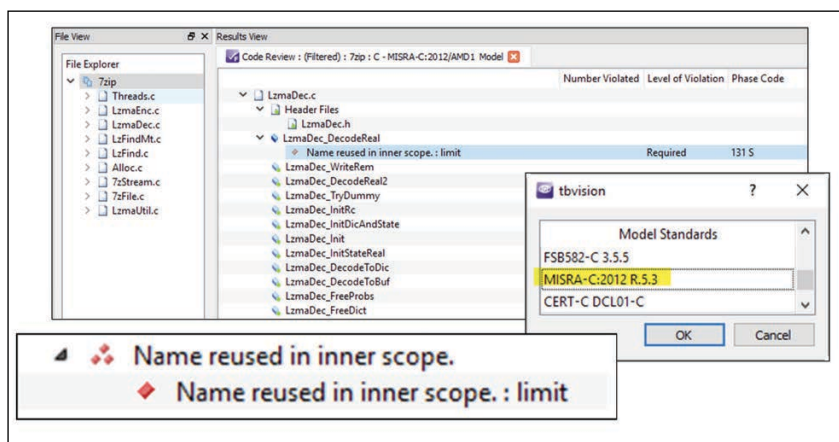


FIGURE 5

This image shows output from LDRA's static analysis tools, as it relates to adherence to a MISRA language subset.

standards—is designed to ensure that developers avoid using those problem features. In addition to showing compliance with coding standards, LDRA static analysis tools can also help developers in many other ways such as by ensuring that their code is clear, easy to maintain and test and not excessively complex.

While static analysis involves an automatic "inspection" of the source code, dynamic analysis involves its compilation and execution either as a whole, or in part. LDRA's unit, integration and system dynamic analysis tools are used to ensure that the code works in accordance with project requirements, and has been exercised adequately. LDRA requirements traceability tools show that the code fulfils the requirements of both the project and any applicable functional safety standards, and that there is no spurious code.

Like some of the other solutions mentioned earlier, LDRA's static analysis, dynamic analysis and requirements traceability tools leverage the benefits of being combined into an integrated tool suite. Some key features offered by the tool suite such as data coupling analysis and control coupling analysis draw upon this integration by leveraging static and dynamic analysis in tandem.

Data coupling analysis can identify issues such as mismatches in the sequences of variable values being set and used, and control coupling analysis can identify problems including ambiguities in the intended control flow of the code. These checks are obligatory for some DO-178C compliant (aerospace) applications and although they might not obligatory elsewhere, that doesn't make the anomalies any less of a threat in other safety- or security- critical systems. Control and data coupling analyses are particularly significant in the context of tainted data, for example, because they point to situations where that data could be inaccurate, and where there is the very real potential for bad actors to abuse the situation. **E**

Product Focus: PC/104 Boards

Legacy That Stacks Up

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

FIGURE 1


PC/104 technology provides the computing inside Klein's UUV-3500 high resolution side-scan sonar system. The system is aboard OceanServer's Iver3 AUV shown here.

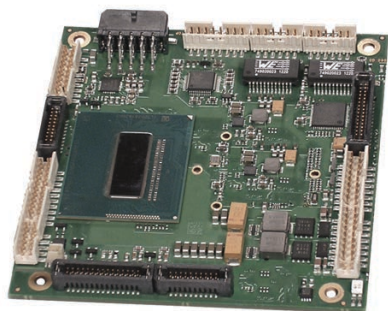
With roots that grew from the ISA-bus era, PC/104 has grown into an embedded board-level form factor suited to the PCI Express landscape. For space-constrained applications, PC/104 and all its follow-on variants continue to meet system design needs.

Since its creation over 25 ago, PC/104 has enjoyed one of the greatest success stories in terms of leveraging technologies from the PC infrastructure. The well-established PC/104 standard is remarkable for opening the door to the embedded stackable computing concept. It began with the ISA bus and over the years has grown to include the latest innovations in desktop computing technologies with PCI and PCI Express. PC/104 evolved through the era of PCI and PCI Express by spinning off its wider family of follow on versions including PC/104-Plus, PCI-104, PCIe/104 and PCI/104-Express.

The PC/104 architecture demonstrates that it's possible to successfully implement quickly evolving PC technology into embedded computing products by taking advantage of PC market adoption, performance, scalability and growing silicon availability worldwide. PC/104 was designed to be simple in design, but rugged in performance. As a result, PC/104 products have permeated many industries. A PC/104 board provides the computing inside Klein's UUV-3500 high resolution side scan sonar for unmanned underwater vehicles. The system is used on OceanServer Technology's Iver3 AUV (Figure 1).

A couple years ago the PC/104 Consortium made a revision to PCI/104-Express and PCIe/104 that provides an additional option called "OneBank". The PCIe/104 OneBank utilizes a smaller, lower-cost bus connector which is compatible to the full size PCIe/104 connector currently in use today. It allows designers to stack boards using a complimentary format that frees up PCB real estate for additional components as well as potential cost savings. The OneBank connector concept consists of removing two of the three "banks" of the standard PCIe/104 connector, resulting in a 52-pin connector as opposed to the full-size 156-pin connector.

Among the more recent trends in PC/104 has been roll out of boards that include Mini PCIe sockets. Mini PCIe lets system designers mix and match add-on functions, leveraging the emerging ecosystem of Mini PCIe peripheral cards as they become available. The product gallery on the next couple pages shows a mix of board designs upgraded to sport the latest processor and memory technologies. These are representative examples of PC/104, PC/104-Plus and PCI/104-Express board-level products. 

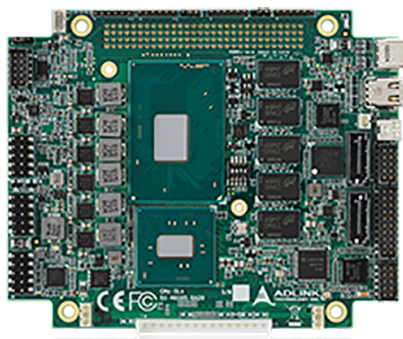


PCIe/104 SBC Boasts Extended Temperature Support

The ADLE3800PC from ADL Embedded Solutions is based on Intel's SoC E3800 Atom product family. The board is well suited for rugged, extended temperature intelligent systems. It has a wide thermal junction temperature (Tj) ranging from -40°C to +85°C. Intel's 7th generation graphics engine on the processor is capable of decoding 10 or more streams of 1080p video and has integrated hardware acceleration for video decode of H.264, MVC, VPG8, VC1/WMV9 and others standards.

- Intel E3800 Series SoC Processors; Dual/quad
- Up to 8 GB DDR3L-1333; 1.35 V SoDIMM204 socket
- CPU TDP 8 W to 10 W
- Type 2 downward-stacking PCIe/104 V2.01
- 2x Gen2 PCIe x1 lanes
- 2x SATA 3 Gb/s; shared with mSATA socket
- 2x 10/100/1000 Mbit Ethernet LAN port
- 2x RS232 COM ports
- 8x USB 2.0 total
- Microsoft Azure certified for IoT

ADL Embedded Solutions
www.adl-usa.com

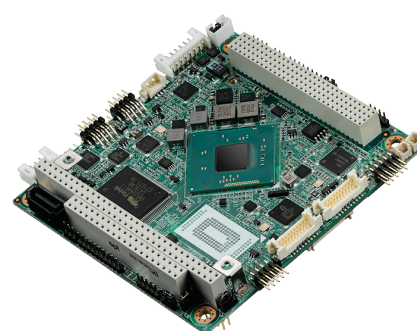


PCI/104-Express Type 1 SBC Sports 6th Gen Core i3

ADLINK Technology's CMx-SLx is a PCI/104-Express Type 1 SBC featuring the 64-bit Intel 6th gen Core i3 processor (formerly "Skylake-H"), supported by the Intel CM236 Chipset. The CMx-SLx is specifically designed for customers who need high-level processing and graphics performance in a long product life solution. The CMx-SLx Intel processor supports Intel Hyper-Threading technology and 8/16 GB of soldered ECC DDR4 memory at 1866/2133 to achieve optimum overall performance.

- 6th gen Intel Core processor (formerly codenamed Skylake)
- Up to 16 GB DDR4-ECC soldered memory
- 3x DDI channels, 1x micro HDMI, 1x mini DP and 1x 18/24 bit single channel LVDS
- 4x PCIe x1 and 1x PCIe x 16 (PEG)
- 2x GbE LAN, 2x SATA 6 Gb/s, 1x USB 3.1, 6x USB 2.0, 2x COM, 8x GPIO
- Supports Smart Embedded Management Agent (SEMA) functions
- Extreme rugged operating temperature -40°C to +85°C variant

ADLINK Technology
www.adlinktech.com



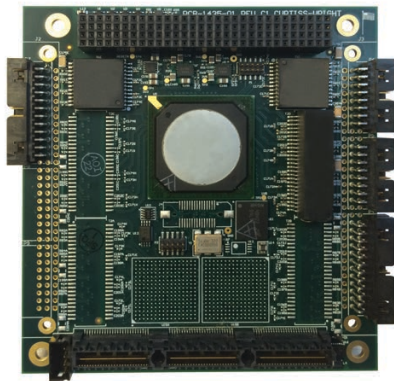
Atom-based PC/104-Plus SBC Boasts Low Power Operation

Advantech's PCM-3365 is a PC/104-Plus SBC with an Intel Atom E3825/E3845/ N2930 processor, supporting DDR3L SDRAM and soldered flash up to 64 GB. PCM-3365 offers an extend temperature SKU with E3825/E3845 SoC. The Thermal Design Power (TDP) rating for the SoC is only 5.7 W for E3825 (the lowest), and 7.7 W for E3845 (the highest). The card is PC/104-Plus form factor which means it supports both ISA and PCI bus through PC/104 and PCI-104 connectors.

- Intel Atom E3825/E3845 and Celeron N2930, DDR3L-1066/1333 SODIMM up to 8 GB
- DirectX11, OpenGL3.2, OpenCL1.1, 3 independent displays: VGA+LVDS/ HDMI+LVDS/ DVI+LVDS/ VGA+LVDS
- Support PC/104-Plus expansion
- 1 Gbit Ethernet, 3x COM, SATA, 6x USB2.0, SMBus/I²C, GPIO, full-size Mini PCIe/full-size mSATA
- Supports SUSIAccess and Embedded Software APIs

Advantech
www.advantech.com

PC/104 Boards

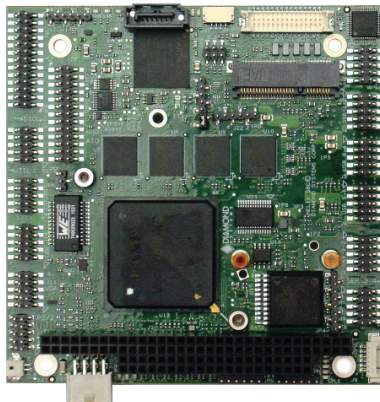


PCI/104-Express Card Provides 20- or 8-Port Gbit Ethernet Switch

The Parvus SWI-22-10 from Curtiss Wright Defense Solutions is a rugged Gbit Ethernet switch card optimized for SWaP sensitive embedded military and civilian computer network systems applications. Featuring advanced Layer 2 networking features with from 8- to 20-ports of 10/100/1000 Mbps connectivity, an integrated management processor, low power consumption, and robust carrier Ethernet software features, the SWI-22-10 enables reliable LAN switching across -40°C to +85°C temperature ranges.

- Rugged embedded Gigabit Ethernet switch
- 20 port and 8 port versions
- Layer 2 fully managed network switch with Layer 3 static routing capability
- Low-power, Energy Efficient Ethernet (802.az) compliant
- IEEE-1588v2 Precision Timing Protocol (PTP) support
- Qual tested to MIL-STD-810 for 40°C to +85°C and high shock/vibration

Curtiss-Wright Defense Solutions
www.curtisswrightds.com

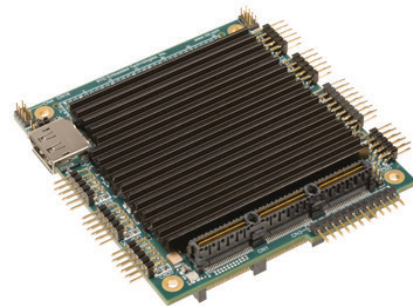


1 GHz Vortex86DX3 PC/104 Board has On-board Data Acq

Diamond Systems' HELIX PC/104 SBC is based on the DMP Vortex86DX3 system-on-chip (SoC) processor. It offers high feature density in a compact size and providing optional integrated high-quality data acquisition circuitry, PCIe MiniCard I/O expansion and rugged construction. Two standard Helix models are available off-the-shelf; one aimed low-cost basic applications and the other targeting data acquisition applications.

- 1 GHz dual core DMP Vortex86DX3
- Up to 2 GB of on-board 64-bit DDR3 SDRAM
- 24-bit LVDS LCD and VGA CRT display support; 1920 x 1080 maximum resolution.
- A broad range of system I/O, including 4 multiprotocol serial ports, 6 USB ports, 2 10/100/1000 Ethernet ports, and 1 SATA port
- PC/104 (ISA) and PCIe MiniCard / mSATA sockets
- Optional data acquisition circuitry: 16 16-bit A/D channels, 4 16-bit D/A channels, and 11 programmable digital I/O lines

Diamond Systems
www.diamondsystems.com

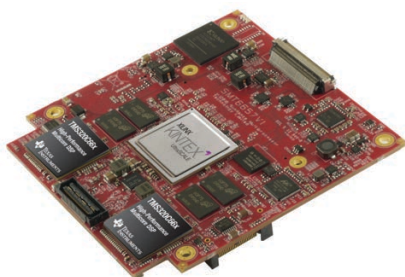


PCI/104-Express SBC Marries 1.9 GHz Atom and 32 GB SSD

The CML24BT from RTD Embedded Technologies is an advanced PC/104 single board computer with a PCI/104-Express stackable bus structure. This Intel Atom based CPU is exceptionally suited for intelligent systems requiring low power consumption in harsh thermal conditions. The surface-mount Type 2 PCI Express connectors enable users to stack multiple peripheral modules above and below the CPU.

- PC/104 form factor, PCI/104-Express stackable bus structure, PCIe Type 2 expansion buses
- Intel Atom E3800 Series Processor; 1.33 GHz, 1.46 GHz and 1.91 GHz options
- Single-Channel DDR3 SDRAM surface-mounted with ECC
- Surface-mounted industrial-grade SATA 32 GB flash drive
- 4 x1 PCIe Links; 1 SATA Port; 4 Serial Ports (RS-232/422/485); 7 USB; Gbit Ethernet; DisplayPort, DVI and HDMI
- -40°C to +85°C standard operating temperature

RTD Embedded Technologies
www.rtd.com

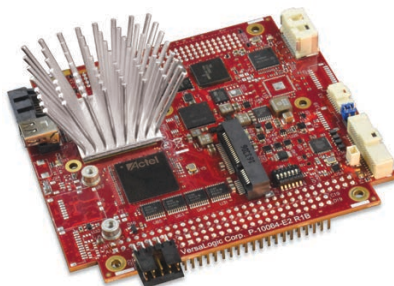


PCIe/104 OneBank SBC with FPGA and Two Dual-DSPs

The SMT6657 DSP+FPGA module from Sundance Multiprocessor Technology is a reliable and flexible platform for DSP applications requiring high-performance integer and floating-point computation. It is applicable to both symmetric multiprocessing applications in which the computational load is shared by the two DSPs and asymmetric applications where one of the DSPs is responsible for hard real-time processing and the other acts as a supervisor.

- PCIe/104 OneBank SBC
- Two TI dual-core 1.24 GHz TMS320C6657 floating-point DSPs
- Xilinx Kintex-7 UltraScale KU35 FPGA
- Serial RapidIO and Hyperlink connectivity between DSPs
- Accepts one VITA57.1 FMC-LPC Mezzanine Card data acq add-on module
- Additional stack-down Serial RapidIO connector to SMT-Carrier-GSI
- Front panel I/O connector carrying Gbit Ethernet and flexible FPGA I/O

Sundance Multiprocessor Technology
www.sundance.com

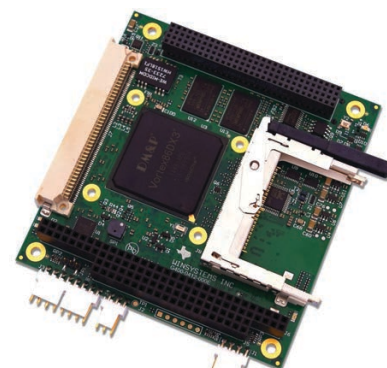


PC/104-Plus SBC Sports Dual Core Bay Trail SoC

The SandCat from VersaLogic is a low-power dual-core SBC with an industry-standard PC/104-Plus expansion interface. This combination enables easy upgrades to existing PC/104 systems to Intel's long-life Bay Trail processor, while preserving plug-in expansion to existing specialty I/O boards. The board also contains on-board I/O interfaces, including USB, a mini PCIe expansion socket and digital I/O ports.

- 1.33 GHz Intel Bay Trail Processor, dual core
- Integrated Intel Gen 7 graphics core supports DirectX 11, OpenGL 4, and H.264, MPEG-2 encoding/decoding. Mini DisplayPort video output
- Up to 8 GB DDR3L DRAM
- Ethernet interface, 2x RS-232/422/485 serial ports; 4x USB 2.0 ports, three 8254 timer/counters, I²C and audio support
- Industry-standard PC/104 and PC/104-Plus expansion
- -40°C to +85°C operation
- MIL-STD-202G qualified for high shock and vibration

VersaLogic
www.versalogic.com



Low Power PC/104-Plus SBC has Rich I/O

Winsystems' PPM-C412 is a PC/104-Plus form factor SBC featuring the latest generation DMP Vortex86DX3 SoC processor. Its small size, low power, rugged design and extended operational temperature make it well suited for industrial IoT applications and embedded systems in the industrial control, transportation, Mil/COTS and energy markets.

- Low Power 1 GHz DMP Vortex86DX3 processor (dual core)
- PC/104-Plus form factor
- 2 GB DDR3-LV System RAM
- 4x USB 2.0 ports, 4x serial ports, Dual Ethernet
- CompactFlash, SATA
- Dual video output (VGA, LVDS with digital backlight dimmer)
- -40°C to +85°C temperature operation

Winsystems
www.winsystems.com

Embedded System Essentials

Attacking USB Gear with EMFI

Pitching a Glitch

Many products use USB, but have you ever considered there may be a critical security vulnerability lurking in your USB stack? In this article, Colin walks you through an example product that could be broken using electromagnetic fault injection (EMFI) to perform this attack without even removing the device enclosure.

By
Colin O'Flynn

In past articles I've taken you through various theoretical attacks on embedded systems, demonstrated various attacks in standard systems and summarized recent work from relevant conferences. This article is something new. I'm going to be presenting a new attack. While it's been disclosed to the vendor—and should have been fixed by the time you read this—you are getting as close to the bleeding edge of attack information as I can present in this article.

Our victim will be a Trezor bitcoin wallet.

This little device can be used to store Bitcoins, which ultimately means a method of securely storing a private key used for cryptographic operations. We don't need to dig into details of the wallet operation, but a critical piece of information to understand is the idea of a "recovery seed". This recovery seed is a series of words which encodes a recovery key, and knowing that recovery seed is sufficient to recover the secret key.

This means someone who steals only that recovery seed—without further access to the wallet—could access funds stored on the wallet itself. It goes without saying that an attack finding that key would be rather detrimental to our experience using the wallet.

It should be noted that there has been some other work that inspired this attack. The "wallet.fail" presentation at the Chaos Communication Congress (CCC) by Dmitry Nedospasov, Josh Datko and Thomas Roth demonstrated how one could break the STMicroelectronics (ST) STM32F2 security protection, allowing the dumping of its SRAM contents. Instead, I'm going to be showing you how to directly dump flash memory where the seed is stored. So, it's a different attack but with similar end results.

I'm going to be using electromagnetic fault injection (EMFI), enabling us to actually perform the attack *without even removing the enclosure*. This means someone can perform the attack without leaving a trace of

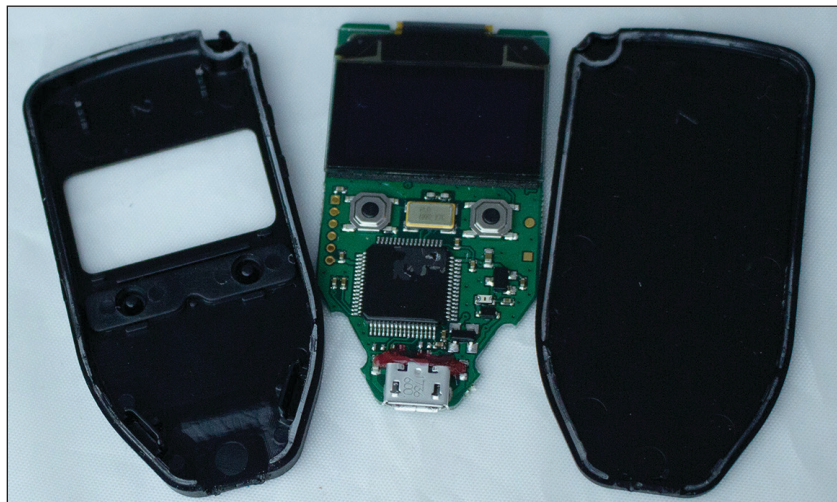


FIGURE 1

The Trezor wallet is shown here with the enclosure removed.

```
#define FLASH_BOOT_START (FLASH_ORIGIN)
#define FLASH_BOOT_LEN (0x8000)

#define FLASH_META_START (FLASH_BOOT_START + FLASH_BOOT_LEN)
#define FLASH_META_LEN (0x8000)

#define FLASH_APP_START (FLASH_META_START + FLASH_META_LEN)
```

modifying the wallet, no matter how carefully you inspect it. Before we get to the real attack, we need to cover some background.

POWERFUL EMFI

EMFI is a powerful method of performing fault injection attacks. Typically, we use some sort of pulse generator to drive an inductor and the inductor will generate a strong magnetic field. If you bring this magnetic field near a chip, this will induce voltages inside metal on the chip. The result is an ability to manipulate internal voltage levels and

insert ringing onto the power bus, causing the device to misbehave. These misbehaving activities are what we refer to as faults or glitches. Such faults or glitches could corrupt data (registers, SRAM) or corrupt program flow.

The Trezor wallet is open-source, which makes this attack a wonderful demonstration to teach you about EMFI and fault injection. You can freely modify the code, program old versions before they patched the bug, and generally perform other useful work to demonstrate this attack.

LISTING 1

memory.h showing *FLASH_META_START* occurs after the bootloader and before the application

```
static int winusb_control_vendor_request(usbd_device *usbd_dev,
                                         struct usb_setup_data *req,
                                         uint8_t **buf, uint16_t *len,
                                         usbd_control_complete_callback* complete) {
    (void)complete;
    (void)usbd_dev;

    if (req->bRequest != WINUSB_MS_VENDOR_CODE) {
        return USBD_REQ_NEXT_CALLBACK;
    }

    int status = USBD_REQ_NOTSUPP;
    if (((req->bmRequestType & USB_REQ_TYPE_RECIPIENT) == USB_REQ_TYPE_DEVICE) &&
        (req->wIndex == WINUSB_REQ_GET_COMPATIBLE_ID_FEATURE_DESCRIPTOR)) {
        *buf = (uint8_t*)&winusb_wcid;
        *len = MIN(*len, winusb_wcid.header.dwLength);
        status = USBD_REQ_HANDLED;
    } else if (((req->bmRequestType & USB_REQ_TYPE_RECIPIENT) == USB_REQ_TYPE_INTERFACE) &&
               (req->wIndex == WINUSB_REQ_GET_EXTENDED_PROPERTIES_OS_FEATURE_DESCRIPTOR) &&
               (usb_descriptor_index(req->wValue) == winusb_wcid.functions[0].bInterfaceNumber))
    {
        *buf = (uint8_t*)&guid;
        *len = MIN(*len, guid.header.dwLength);
        status = USBD_REQ_HANDLED;
    } else {
        status = USBD_REQ_NOTSUPP;
    }

    return status;
}
```

LISTING 2

The function *winusb_control_vendor_request* from *winusb.c* responds to requests for various information related to *WinUSB* over the control USB endpoint. Note the call "*MIN(*len, guid.header.dwLength)*" which decides on the length of the returned response.

You can see the sources for Trezor on github. See the *Circuit Cellar* article materials webpage for the specific github link. If you want to follow this article, be sure to select the “v1.7.3” tag on GitHub. These flaws are fixed in a firmware release that will be available by the time you read this article, so you should look at the older (vulnerable) code to better understand the exact attack. The Trezor is based on ST’s STM32F205 and you can see with Trezor sans enclosure in **Figure 1**. Note that the STM32F205 is just below the surface of the enclosure—a feature we will use to improve our attack.

The actual sensitive recovery seed is stored in flash memory. It’s located just after the bootloader, as shown in **Listing 1**. The bootloader can be entered by holding down the two buttons on the front of the Trezor, and allows a firmware update to be loaded over USB. Since a malicious firmware update could simply read out this flash location, the bootloader will verify that various signatures are present on a firmware update to prevent such an attack. Loading unverified firmware would be one method of attack, but isn’t what we are going to use. The problem with all of these attacks is that the design of the Trezor erases the flash memory *before* loading and validating the new file, storing the sensitive metadata in SRAM during this process. The wallet.fail disclosure actually attacked this, since it’s possible to glitch the STM32 to go from code read protection level RDP2 (which

completely disables JTAG) to level RDP1 (which enables JTAG to read from the SRAM, but not from the code).

If our attack corrupted the SRAM—or needed a power cycle to recover from error states—performing that erase is very dangerous. The wallet.fail attack was able to recover the SRAM, but the attack method we will use could corrupt the SRAM. That means any mistake would permanently destroy the recovery seed. Instead, we are going to try and directly read out the flash memory. This is much safer since we never perform an erase command, meaning the data is safely stored in memory waiting for us to extract it.

USB READ REQUEST

Because the bootloader contains USB, it also contains very standard USB processing code. Part of this is shown in **Listing 2**, which comes from the file `winusb.c`. I’ve chosen this particular request because there are actually two data structures present that are returned by this code—one is stored in FLASH and one is stored in SRAM. The USB request being processed first checks some information sent about the request. It looks for a matching `bRequest`, `bmRequestType` and `wIndex` which are all attributes of a USB request. Finally, the USB request itself contains a `wLength` field, which is how much data the computer is requesting be sent back. I can freely request up to `0xFFFF` bytes of data—and that is exactly what I will do. But, as you can see, the code does a `MIN()` operation to limit the length of the actual data sent back to be the minimum of either the requested length or the size of the descriptor I will send back.

So, what happens if that check was wrong? While it would let me send back the descriptor, along with all the 64K (`0xFFFF`) bytes of data that lies after the descriptor itself. This includes our precious metadata—the USB stack simply sends back the block of data as the computer requested. The entire security of the system depends on one simple length check!

If you’ve read a few of my articles, you might guess I’ve got a plan. We will be using fault injection to bypass the check that depends on a single instruction. Before we dive into details of performing the actual fault, let’s do a bit of “sanity check” on my claims. You can use these sanity checks in your own code to help understand the impact of similar vulnerabilities.

DISASSEMBLING CODE

The first sanity check is to confirm that a simple fault model can cause our intended operation. This can be trivially confirmed by inspecting a disassembly of the code, done

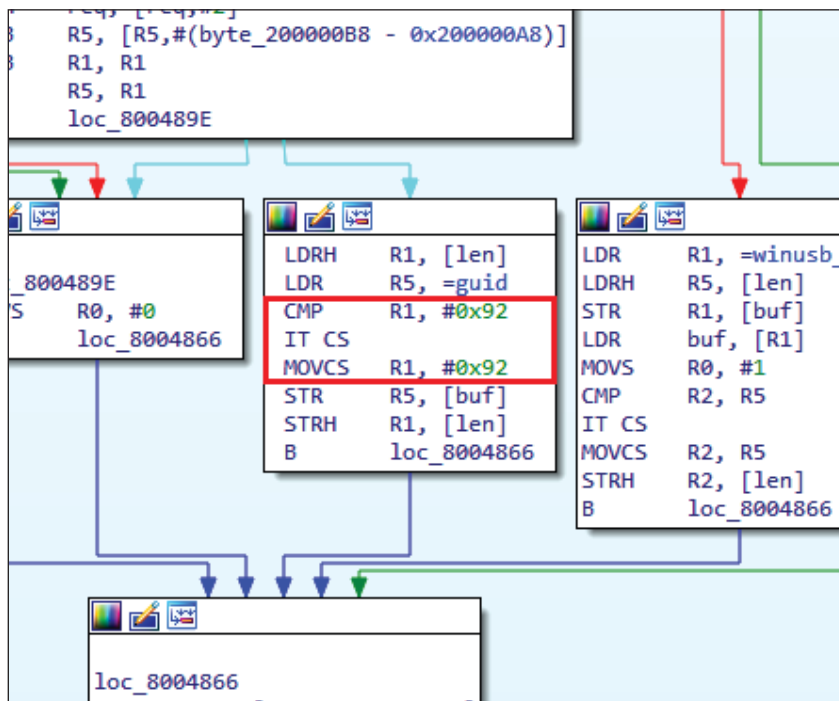


FIGURE 2

IDA disassembly of the function in question ultimately shows a single assembly instruction separates your sensitive data from being politely sent back on the USB port.

Index	m:s.ms.us.ns	Len	Err	Dev	Ep	Record	Summary
0	0:00.000.000.000					● Capture started (Aggregate)	[02/06/19 00:45:55]
1	0:00.000.000.000					🚩 <Host connected>	
2	0:00.000.633.500					🚩 <Full-speed>	
3	0:23.658.183.950	146 B		22	00	▶ 📁 Control Transfer	92 00 00 00 00 01 05 00 01 00 8
24	0:06.791.576.583	146 B		22	00	▶ 📁 Control Transfer	92 00 00 00 00 01 05 00 01 00 8
45	0:03.879.450.166	146 B		22	00	▶ 📁 Control Transfer	92 00 00 00 00 01 05 00 01 00 8
66	1:58.972.722.583	65535 B		22	00	▶ 📁 Control Transfer	92 00 00 00 00 01 05 00 01 00 8
4171	0:11.333.695.616					● Capture stopped	[02/06/19 00:48:40]

FIGURE 3
Using a debugger to step over the single check (or recompiling the code) shows that large chunks of memory will be sent back on request.

with IDA in **Figure 2**. Note in particular that due to the resulting code flow, we need to skip only a single instruction to accomplish our goal of having the user-supplied length field be accepted.

The second sanity check will be to confirm there is not some higher-layer protection. For example, maybe the USB stack does not actually accept such a large response given that there's no actual need for this? This is a little harder to prove by simple inspection, but the open-source nature of the Trezor makes this possible. What we can do is modify the code to simply comment out the security check. If you didn't want to recompile the code, but did have debugger access, you

could also use an attached debugger. Use the debugger to set a breakpoint before the new value is copied over and toggle the status of the flag, or manipulate the program counter to bypass the instruction.

Validating this sanity check will be done in the same way as the actual attack. This will use the code from Listing 2. This code sends the WinUSB control request which should return with the guid structure. It sends a length request of 0xFFFF for the request, which should be paired down to 146 bytes by the code. As you can see from **Figure 3**, when I do not modify the instruction, the USB request results in the expected-size response. Modifying the instruction (or using

COLUMNS



LOOKING TO
**ADVANCE
YOUR
CAREER?**

Be a part of one of the
**top Electrical Engineering
programs in country**
and experience the
Bearcat Promise!



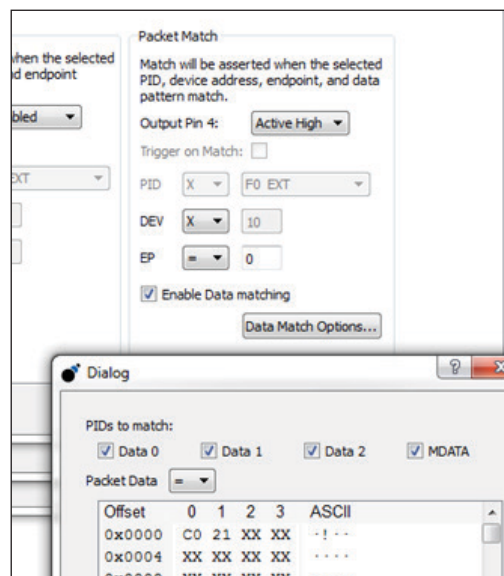
University of
CINCINNATI | ONLINE

online.uc.edu

Fall registration is
open now

FIGURE 4

The USB protocol analyzer is setup to trigger on a specific packet related to our request.



a debugger to manually clear the comparison flag) to bypass this check results in a full-size response. This demonstrates that there is no “hidden feature” that will fundamentally prevent the attack from working. With that knowledge, let’s move onto getting this thing talking to us!

USB TRIGGERING AND TIMING

Before we can talk about how we insert the glitch, we need to know where to insert the glitch. We do know the exact code that triggers the glitch, and we do know the command we sent over USB. But we need to get better than that to introduce the exact instruction. In my case, since I have access to the software I’m going to “cheat” during my first test and measure the actual execution time. If I didn’t have this capability, I would end up with a much slower sweep of possible locations.

The first thing I’ll do is get a more solid trigger on the USB data itself. The entire area of using USB for glitch triggering was actually started by Micah Scott, who demonstrated voltage glitching to dump the firmware from a drawing tablet and developed a simple module to perform real-time glitching (which she called the FaceWhisperer). Instead I’m

going to use a Total Phase Beagle 480, which can perform triggering based on physical data going over the USB line. The setup for that is shown in **Figure 4**. The Total Phase Beagle 480 also has a beautiful sniffer interface, so I can sniff the traffic and better understand what malformed packets are coming back. This capability is very useful since I can see, for example, the exact portion of the USB request being interrupted/corrupted. That might give me some hints about how far into the code the program has executed.

Besides FaceWhisperer and the Beagle 480, there are other methods of triggering the glitch. Great Scott Gadgets offers its GreatFET device that has a module called GlitchKit. GlitchKit provides similar triggering capabilities, but generates the requests from the GreatFET itself. As of this writing the GlitchKit has more limited response capability, so I wasn’t able to read the entire response back. Finally, you could look into a simple circuit using a USB PHY—such as Microchip Technology’s USB3500—and an FPGA. Watch for the future open-source PhyWhisperer-USB from NewAE Technology which will give you that capability.

Once we have a trigger based on the USB request going “over the wire”, we can insert a trigger by setting an I/O pin high when the sensitive code runs. We use this for characterizing the system, since we can use an oscilloscope to measure the time from the USB packet going over the wire to the sensitive code operating. In this case, the time ends up being around 4.2 μ s to 5.5 μ s. It’s not perfect timing, because there appears to be some jitter due to the USB packets being processed by a queue. We have just learned that, when performing the fault injection demo, we should expect that we do not achieve perfect reliability.

GLITCHING THROUGH THE CASE

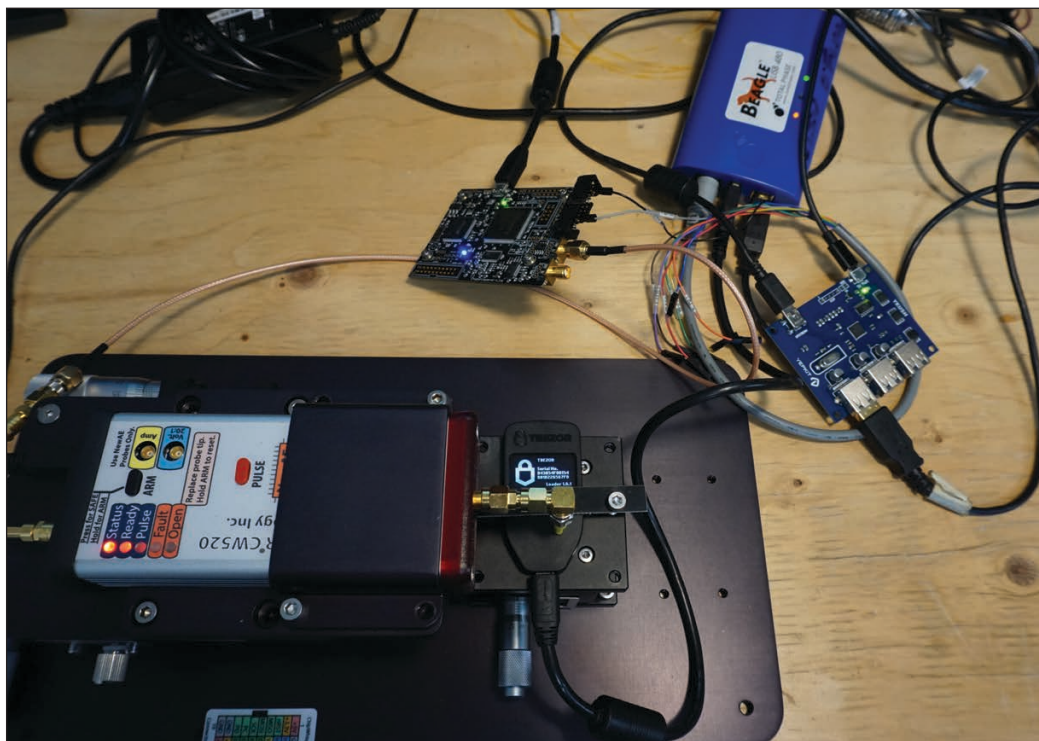
For inserting the glitch, I’m using a setup as shown in **Figure 5**. This includes a ChipSHOUTER EMFI platform, a manual XY table for positioning the coil, the Trezor target, the Beagle 480 to generate a trigger, a ChipWhisperer to generate the timing offset and a Yepkit USB hub which provides a simple API to power cycle attached devices. The power cycle capability is useful as we will be very frequently crashing the target device.

A very simple script (shown in **Listing 3**) enables me to power-cycle the device and issue the WinUSB request. The physical “jig” that holds the Trezor actually holds the two power buttons down, ensuring it always enters bootloader mode on start-up. We want to use the bootloader since the bootloader is at a lower address than the metadata,

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

RESOURCES

Great Scott Gadgets | www.greatscottgadgets.com
 Microchip Technology | www.microchip.com
 NewAE Technology | www.newae.com
 STMicroelectronics | www.st.com
 Total Phase | www.totalphase.com
 Trezor | www.trezor.io
 Yepkit | www.yepkit.com

**FIGURE 5**

Complete setup of the EMFI attack including Beagle 480 for trigger generation, ChipWhisperer for timing modifications, ChipSHOUTER for EMFI insertion and a USB hub to power cycle the target.

so dumping any memory from within the bootloader is more useful when it comes time to recover the metadata.

The success rate is low—less than 0.1% of glitches are successful. We can however achieve a successful glitch within about 1-2 hours on average, making it a relatively useful attack in practice. A successful glitch is one where the USB request comes through with the full length of data, since I was able to bypass the length check. Finding the exact location takes some experimentation—you will get many system crashes due to memory errors, hard faults and resets. But if you are using a hardware USB analyzer such as the Beagle 480 you can see where these errors are happening, which helps you understand the glitch timing. If we didn't have the inside knowledge of the I/O pin we could toggle, this would be very valuable.

Figure 6 shows such an example. Note the USB transaction when performed correctly has a few steps. The upper part of that figure shows a number of correct 146-byte control transfers. The first part is the SETUP phase. The Trezor has ACK'd the SETUP packet, but then never sends the follow-up data. The Trezor entered an infinite loop as it jumped to one of the various interrupt handlers for error detection. As the location of the fault is shifted along in time, various effects on the USB traffic are observed: moving the glitch earlier often prevents the ACK of the setup packet, moving the glitch later allows the first packet of follow-up data to be sent but not the second, and moving the glitch much later

allows the complete USB transaction but then crashes the device. This knowledge helps me understand which part of the USB code the fault is being inserted into, even if that fault is still a sledgehammer causing a device reset instead of an intended single instruction skip.

The final step of fine-tuning the fault to get a useful effect again is helped with our protocol analyzer. I physically moved the coil around over the surface, along with adjusting the glitch width and power level. It was possible—from the LCD screen—to visually see when the device entered an error handler or seemed to continue unaffected. Finding a location that did not always enter an error is typically a useful starting point, and from there I searched through various parameters until a successful glitch occurred. Again, note that due to the deterministic nature of the glitch timing, you must be careful to search sufficiently long in possible candidate glitch settings.



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 NewE Technology both based in Halifax, NS, Canada. Some of his work is posted on his website at www.colinoflynn.com.

```

import time
import time
import usb
import usb.core
import chipwhisperer as cw

def get_winusb(dev, scope):
    """WinUSB Request is most useful for glitch attack"""
    scope.io.glitch_lp = True #Enable glitch (actual trigger comes from Total Phase USB Analyzer)
    scope.arm()
    resp = dev.ctrl_transfer(int('11000001', 2), ord('!'), 0x0, 0x05, 0xFFFF, timeout=1)
    resp = list(resp)
    scope.io.glitch_lp = False #Disable glitch
    return resp

def reset_trezor():
    """Requires a YK USB Hub - has power control of each port"""
    subprocess.check_output([r'ykushcmd.exe', '-d', '1'])
    time.sleep(0.5)
    subprocess.check_output([r'ykushcmd.exe', '-u', '1'])
    time.sleep(1)

# ChipWhisperer used for trigger delay only
scope = cw.scope()
target = cw.target(scope)

# Values found from sweeping around
scope.clock.clkgen_freq = 147E6
scope.adc.basic_mode = "rising_edge"
scope.adc.samples = 500
scope.glitch.clk_src = "clkgen"
scope.glitch.output = "enable_only"
scope.glitch.trigger_src = "ext_single"
scope.glitch.repeat = 1
# Original extclock was 100MHz, so we scale offset
# relative to our actual clock to maintain 4.4uS
scope.glitch.ext_offset = 440
scope.glitch.ext_offset = (scope.glitch.ext_offset / 100.0E6) * scope.clock.clkgen_freq

dev = None

#Loop until we get too large a response
while True:
    if dev is None:
        dev = usb.core.find(idProduct=0x53c0)
        dev.set_configuration()

    try:
        #Perform USB request - glitch trigger happens via
        # TotalPhase Beagle 480
        res = get_winusb(dev, scope)
        if(len(res)) > 146:
            print("Data Over-Run Detected - DONE")
            break
    except usb.USBError:
        reset_trezor()
        res = None
        dev = None

f = open("outputresults.bin", "wb")
f.write(bytearray(res))
f.close()

```

LISTING 3

Shown here is a complete attack script in Python, which sends the USB requests while inserting faults.

FS	6546642	0:00.056.668.166	146 B	28	00	Control Transfer	92 00 00 00 00 01 05 00 01 00 88 00 00 00 07 00 00 00 2A 00 44 00 65 00...	
FS	6546643	0:00.000.000.000	8 B	28	00	SETUP trn	C1 21 00 00 05 00 FF 1A	
FS	6546647	0:00.000.025.333	64 B	28	00	IN trn	92 00 00 00 00 01 05 00 01 00 88 00 00 00 07 00 00 00 2A 00 44 00 65 00...	
FS	6546651	0:00.000.070.750	64 B	28	00	IN trn	00 00 7B 00 30 00 32 00 36 00 33 00 62 00 35 00 31 00 32 00 2D 00 38 00...	
FS	6546655	0:00.000.071.083	18 B	28	00	IN trn	39 00 64 00 38 00 65 00 66 00 35 00 7D 00 00 00 00 00	
FS	6546659	0:00.000.026.333	0 B	28	00	OUT trn		
FS	6546663	0:00.025.202.500	8 B	T	28	00	SETUP trn	C1 21 00 00 05 00 FF 1A
FS	6546664	0:00.000.000.000	3 B	28	00	SETUP packet	2D 1C B8	
FS	6546665	0:00.000.003.416	11 B	28	00	DATA0 packet	C3 C1 21 00 00 05 00 FF 1A 83 9D	
FS	6546666	0:00.000.008.666	1 B	28	00	ACK packet	D2	
FS	6546667	0:00.000.013.166	1.99 s	28	00	[41215 IN-NAK]	[Periodic Timeout]	
FS	6546668	0:02.000.005.333	1.99 s	28	00	[41201 IN-NAK]	[Periodic Timeout]	

PREVENTING THE ATTACK

While it's all good to cause the attack, how would you prevent against it? The first thing is to evaluate if your USB stack can be modified to prevent sending such large responses. If you never need to perform transfers of more than say 256 bytes, why not use an 8-bit number internally, or mask off the upper bits? Such a mask can be applied at multiple locations to complicate glitch attacks.

The second easy fix is to take advantage of memory protection, if your specific device supports it. This fault saw me slide from the USB descriptors in flash memory and read beyond them into sensitive metadata. But if we had bounded the sensitive metadata with invalid memory segments, our "slide" would have caused an exception due to the memory access error. When storing sensitive data in memory—either flash or SRAM—, bounding

it with traps can be useful to catch any sort of attack that reads beyond an array. More generic countermeasures to fault attacks can also be applied, but I wanted to concentrate on specific countermeasures relevant to the memory ready attack shown here.

USE THE (MAGNETIC) FORCE


I hope you enjoyed this case study on electromagnetic fault injection. I've taken you through how EMFI could be used to attack a real product, with an exploit that has recently been disclosed to the Trezor team. Many other USB stacks use an almost identical code flow however, so I suspect you'll find this vulnerability could exist in your own system. Ultimately it depends on the use-case, but anything where sensitive data is stored in standard internal memory needs great care to keep that data inside your device. 

FIGURE 6

A physical USB analyzer (compared to attempting to use a software-only solution) is critical to see mangled packets on the bus, which lets us understand how far into requests the target got before freezing.



Mentor
A Siemens Business

Speed up your PCB design verification by up to **90%**

7 Habits of Highly Efficient PCB Designers

Design habits that expedite design completion, improve design quality, and enhance productivity are instrumental to highly efficient PCB design.

Learn what you can do to succeed!
Learn More in this FREE White Paper
www.circuitcellar.com/mentor



Sales@ezpcb.com

EzPCB
Welcome
www.ezpcb.com
One-Stop PCB & PCBA Turnkey Service

PCB Online Calculator
No Need to Register
Instant Quote & Pay

1 To 40 Layers
Prototype to Mass Production
Amateur to Professional

Prototype Start At \$5/PC
2L 4"x4" each
Free Shipping

In the past years, our PCB have been shipped to 40 countries

The Consummate Engineer

Pressure Sensors

Terminologies and Technologies

By
George Novacek

Over the years, George has done articles examining numerous types of sensors that measure various physical aspects of our world. But one measurement type he's not yet discussed in the past is pressure. Here, George looks at pressure sensors in the context of using them in an electronic monitoring or control system. He looks at the math, physics and technology associated with pressure sensors.

Great efforts are expended by numerous R&D laboratories on development of new sensors capable of detection of just about any physical aspect of our world. After all, sensors are what give systems their intelligence. An important physical quantity whose measurement we have not yet discussed in the past is pressure. In this article I'll look at sensors capable of providing electrical output signal so that it can become a part of an electronic monitoring or control system.

By definition a pressure sensor is a transducer whose purpose is to measure pressure of gases or liquids. A gas or a liquid pressure is equal to the force required to stop that gas or fluid from expanding. It is expressed as a force per unit area.

PHYSICS AND UNITS

Let's start with the fundamental physics. The SI (metric) system designates pressure as a derived unit called Pascal (Pa), named after the French mathematician and physicist Blaise Pascal (1623–1662). The pressure of 1 Pa represents the force of one Newton (N) exerted per one square meter (m²) area. In the metric system, Pa, as the measure of gas or liquid pressure, is frequently substituted by units called atmosphere (atm) or a Torr—where 1 atm = 101,325 Pa = 760 Torr. In the industry 1 atm is often considered to be a reference pressure.

A bar is a metric unit of pressure equaling to exactly 100,000 Pa. That said, bar hasn't been approved by the International System of Units for use as a bona fide metric unit. A bar is slightly less than the average atmospheric pressure on Earth at sea level. A common unit of pressure used in North America is PSI, which stands for "pounds per square inch" and is equivalent to 6.894×10^3 Pa in SI units. In North America you always can encounter a unit referred to as PSIA, which represents the absolute pressure in pounds per square inch relative to vacuum—as opposed to the atmospheric pressure at sea level, which is 14.7 PSIA = 1 bar. Similarly, the PSIG (PSI gauge) designation indicates that the atmospheric pressure is included in the measurement.

Torr is a unit of pressure named in honor of the 17th century Italian mathematician and physicist Evangelista Torricelli (1608–1647). Torricelli is the inventor of the Mercury (Hydrargyrum - Hg) barometer. The principle of the Torricelli barometer is shown in **Figure 1**. Atmospheric pressure acting on a pool of Mercury in a vessel causes the Mercury to rise inside an evacuated tube to a height corresponding to the atmospheric pressure. This is typically 760 mm at the sea level but it also depends on the temperature and altitude. In fact, barometric pressure has been commonly used to establish altitude. Typically, an altitude of an object is:

$$h = \left(1 - \left(\frac{P}{P_{\text{ref}}} \right)^{0.190284} \right) \times 145366.45 \text{ ft.}$$

This measurement is quite accurate up to about 11,000 m (36,090'). Altitude measurement even with an inexpensive barometric sensor [1] can achieve resolution of about 0.3 m (approximately 1')—better than most GPS systems. In general, however, just remember that “normal” barometric pressure is around 760 mm Hg, that is 760 Torr or 1 atm.

I have fond memories of my high school days when building the Torricelli barometer was the first experiment we conducted in the physics class. Those were the “good old days” when the teacher didn’t mind us splashing our bare hands in Mercury in an open vat. Fortunately, those days are over—but the barometer worked and I never forgot how and why.

A millimeter of mercury is also used as a manometric unit of pressure. Most of us are familiar with blood pressure monitors, although most modern instruments use electronic transducers rather than a column of Mercury. The unit was originally defined as the required pressure to raise a column of Mercury by 1 mm, but the definition has been changed to exactly 133.322387415 Pa. It is denoted by the symbol “mmHg.” As one might expect, the inch-of-Mercury pressure is also used and can still be found in aviation and some industries in North America.

PRESSURE-BASED ELECTRONICS

Confused? Even though we’ve been talking the same physical quantity, numerous units and methods of measurement have been used over the time based on history, convenience or application. And there are more—we just don’t have the space to discuss them all here. If you are interested, do your research. There are many articles on the Internet explaining different pressure gauges, units, their conversion and practical use.

All those details are secondary for the engineer faced with a task of designing a pressure-based electronic system. The sensor type, its specification and electrical interface should—and usually is—selected by the system designer and included in the system specification. The circuit designer just has to make sure the specification and especially the units of pressure are correct.

Pressure sensors, whether we call them transducers, gauges, indicators or something else have many uses in automatic control. Besides direct measurement and control

of pressure their outputs can be used to determine altitude, tire pressure, liquid level, fluid or gas flow speed and many others.

Two subcategories I should mention at this time are similar to ones I mentioned with many other transducers. Many transducers can be divided between sensors and switches. Sensors provide continuous analog or digital signal in some defined way proportional to the magnitude of the measured quantity. Switches, on the other hand, generate a discrete on/off signal when a specific magnitude threshold has been reached. Since the switches are primarily just sensors equipped with some kind of a threshold detecting logic, we can concentrate on sensors only.

When selecting a pressure sensor, you need to consider a number of characteristics. Pressure range, operating temperature, the type of pressure, liquid or gas, size, cost, output signal and others are among the most obvious ones. Do we need an absolute

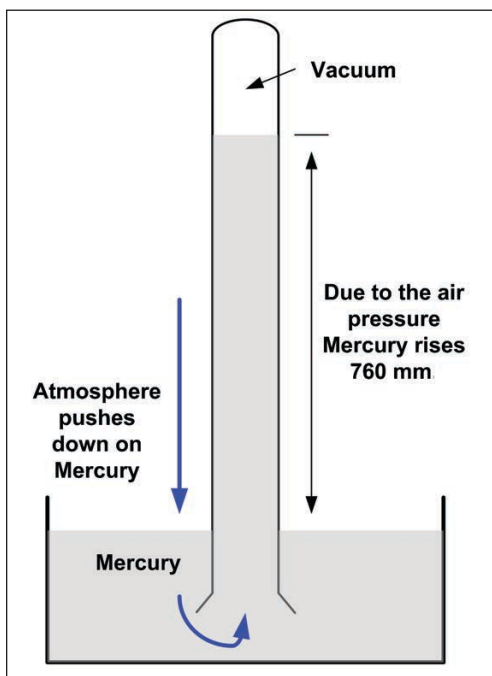


FIGURE 1
The principle of the Torricelli’s barometer

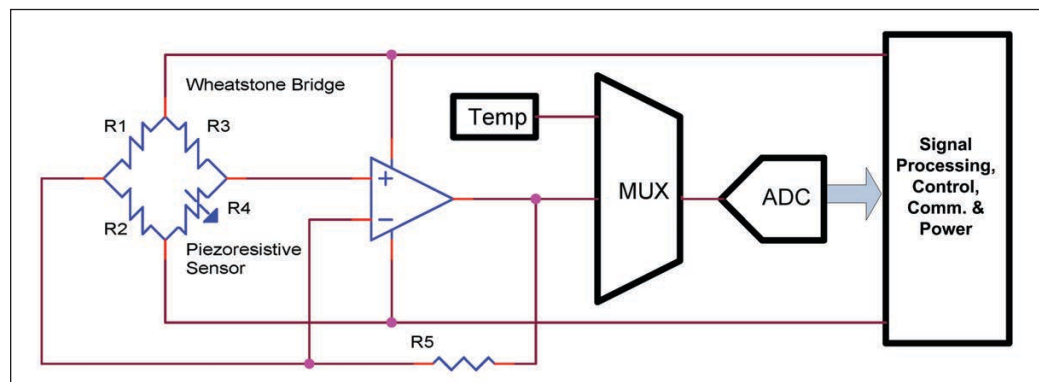


ABOUT THE AUTHOR

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 has contributed articles to *Circuit Cellar* since 1999, penning more than 120 articles over the years.

FIGURE 2

Principle of MPL115A2 operation



or relative measurement? Absolute sensors provide pressure measurement with respect to perfect vacuum. Relative pressure gauges measure pressure with respect to the existing atmospheric pressure. The relative measurement can be both above or below the atmospheric pressure. In the latter case we usually call such transducers vacuum sensors. If you need to measure differential pressure—such as occurring across pumps, blowers, filters and so forth—differential pressure transducers will do the job.

PRESSURE TRANSDUCERS

Pressure transducers can be divided into two basic categories. Force collector and special. Force collector transducers use some mechanical arrangement to convert the pressure acting on a known area into a movement, displacement, strain or a distortion of a mechanical component which can be subsequently converted into an electrical signal. Typical examples would be diaphragms, pistons, bourdon tubes, bellows

and others. Various technologies are used to convert the results of the mechanical movement or strain into electrical, usually analog signal, which can be and quite often is these days, digitized. Special type transducers are, as their name suggests, not very common. Some rely on resonant frequency changing with pressure, thermal conductivity, ionization stream and so forth.

Strain or displacement conversion methods are similar to the ones I described in my previous articles on transducers, such as “Accelerometers Revisited” (*Circuit Cellar* 334, May 2018) [2]. Piezoresistive strain gauge is a popular technology, commonly employed for general purpose measurement. However, it is sensitive to temperature and, therefore requires appropriate compensation. Consequently, piezoresistive transducers such as NXP Semiconductors’ MPL115A2 must contain a temperature sensor as well.

Internally, the strain gauge is usually a part of a Wheatstone bridge (**Figure 2**). Its resistance increases with the increasing strain. Capacitive sensors rely on a diaphragm being a part of a variable capacitor. Here, the capacitance usually decreases with the rising pressure. Many other methods of displacement or distortion detection are used—some of which I described in my previous articles. Among these are LVDT (linear variable differential transformer), inductance change Hall Effect and others. Imagination has no limits. Piezoelectric effect—due to its very nature—makes piezoelectric sensors unsuitable for measurement of static forces and relegates those transducers to dynamic measurements.


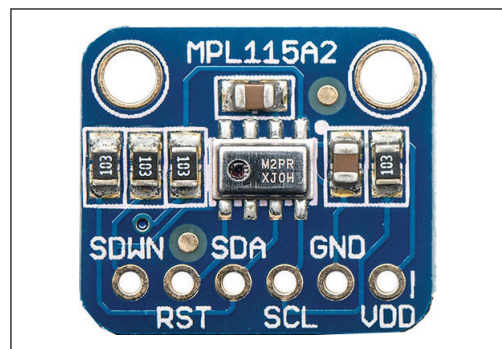
In practical terms, it is easy to experiment with pressure measurement. Various pressure sensors on break-out boards can be purchased from vendors such as Adafruit [1], SparkFun [3] and others for literally just a few dollars. Most break-out boards, such as the MPL115A2 I²C Barometric Pressure/Temperature Sensor Board in **Figure 3** are available with I²C interface and can be readily used with platforms such as Arduino. Order one of these break-out boards and have fun! 

FIGURE 3

Barometric pressure/temperature sensor board with NXP MPL115A2 transducer and I²C interface



For detailed article references and additional resources go to: www.circuitcellar.com/article-materials
Reference [1] through [4] as marked in the article can be found there.

RESOURCES

Adafruit | www.adafruit.com
NXP Semiconductors | www.nxp.com
Sparkfun | www.sparkfun.com

sensors JUNE 25-27 expo & conference **2019**

MCENERY CONVENTION CENTER / SAN JOSE, CA

The industry's largest event dedicated to

SENSORS, CONNECTIVITY, AND SYSTEMS.

Sensors Expo & Conference is where you'll find the best of the best in the sensors industry, along with new and innovative ways to jump start your sensor solutions. Be a part of the ONLY event where technologists find opportunities and engineers innovate solutions. This year's event features three exciting days of all-new Pre-Conference Symposia, Conference Technical Sessions across 10 updated tracks, visionary Keynote Presentations, Networking Events, Exhibits, and much more!



7,000+ ATTENDEES // **330+** EXHIBITORS // **100+** SPEAKERS //
65+ CONFERENCE TECHNICAL SESSIONS // **5** PRE-CONFERENCE SYMPOSIA // **2** KEYNOTES

REGISTER TODAY!

Use code **CC100** for \$100 off Conference Passes
or a FREE Expo Hall Pass!

OFFICIAL PUBLICATION

sensors
ONLINE

INDUSTRY SPONSOR



CO-LOCATED WITH



WWW.SENSOREXPO.COM #SENSORS19



Picking Up Mixed Signals

Fancy Filtering with the Teensy 3.6

Arm-ed for DSP

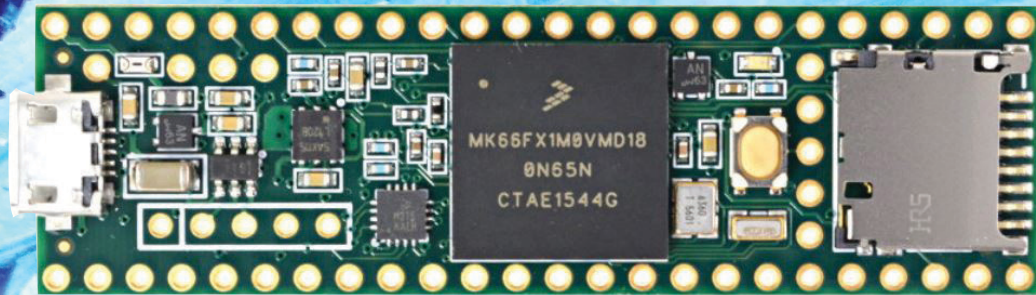


FIGURE 1

Top view of the Teensy 3.6 Arm MCU module. To the right is the on-board MicroSD socket, which accepts the MicroSD card containing the Cabinet Impulse Response file.

Signal filtering entails some tricky tradeoffs. A fast MCU that provides hardware-based floating-point capability eases some of those trade-offs. Here, Brian has used the Arm-based Teensy MCU modules to serve those needs. Here, Brian taps the Teensy 3.6 Arm MCU module to perform real-time audio FFT-convolution filtering.

By
Brian Millier

Signal filtering can be done either with analog circuitry or digitally using a microcontroller (MCU) coupled with analog-to-digital and digital-to-analog converters. The strength of analog filters is that they can cover wide frequency ranges. If they are designed entirely with passive components, the range of signal amplitudes that can be handled is limited only by the voltage rating of the various capacitors that are used. Additionally, they don't add much, if any, noise to the signal. However, a limitation of analog filters is that they can't provide a sharp cut-off rate at their corner frequency (F_c), unless you cascade many filter sections and use close-tolerance components.

If you need high-performance filters, then digital filters might be the way to go. You can design very sharp low-pass, high-pass, notch and band-pass filters using digital techniques, if you use high-resolution ADC/DACs to convert the analog signal into the digital domain and (optionally) back to the analog domain. However, the MCU that you use must be fast and, in general, feature hardware-based floating-point operations. Two years ago, I discovered a line of Arm-based MCU modules that fill the bill nicely.

In *Circuit Cellar* issues 324 (July 2017) and

325 (August 2017), I described a digital guitar amplifier based upon the Teensy 3.2 Module, which contains an Arm Cortex-M4 MCU. The analog guitar signal was converted to a 16-bit digital signal for processing, and then back to an analog signal for power amplification, by an NXP Semiconductor SGTL5000 Codec contained on the PJRC Audio Shield. This project was made possible largely due to the extremely powerful Audio library provided by the manufacturer of the Teensy modules. This library consists of many audio functions, all of which operate using DMA transfers and interrupt service routines (that is, as a background task). The sampling is done at CD quality (44,100 samples/s at 16-bit resolution).

That project involved many different audio functions—some from the Teensy Audio Library, and some that I wrote myself. The filtering I used for the project was in the form of a 5-band parametric equalizer (EQ). This consists of five blocks of band-pass filters, each one centered on a specific frequency in the audible range. Such an EQ is basically a sophisticated “tone control” for the guitar signal. While most of the other guitar signal processing was done within the Teensy 3.2 MCU, using the Audio library, the 5-band parametric EQ was handled by a DSP block

contained within the SGTL5000 Codec on the Teensy Audio Shield.

After finishing that project, I became interested in more sophisticated filtering algorithms that could be performed by the Arm MCU found on the Teensy modules. The Teensy Audio Library routines work with all the Arm-based MCUs in the Teensy module family (except the lowest-cost LC model). The Audio library contains three types of digital filters:

- 1) Biquad (low pass, high pass, band pass, notch)
- 2) FIR (up to 200 taps)
- 3) State-variable (Chamberlin)

The Biquad algorithm executes quickly, and its coefficients are easy to calculate on the fly, which makes it easy to change the filter bandwidth and F_c quickly. Finite impulse response (FIR) filters can provide much better filter characteristics, if you configure them with enough “taps”. However, as you increase the number of taps used, the execution time increases proportionately.

All the above filters use 16-bit, fixed-point math (Arm Cortex M4 DSP instructions using the Q15 data format). This is fast and reasonably accurate, but not enough to provide very sharp filter “skirts”. When you attempt to cascade several sections of such filters, you start to see the limitations in the precision of the fixed-point math.

The higher-end Teensy modules (Teensy 3.5 and 3.6) contain the more powerful Arm Cortex M4F core. These devices have hardware floating-point instructions, which basically allow you to do floating-point operations as quickly as you could do the 16-bit fixed-point operations with the DSP instructions available on Teensy 3.2’s Arm Cortex M4 MCU.

By using a Teensy 3.6 with hardware floating-point instructions, I figured that I could handle more sophisticated filtering algorithms. Another consideration was that the Teensy 3.6 MCU runs at 180 MHz, compared to the 72 MHz clock speed of the Teensy 3.2. Also, the Teensy 3.6 can be safely over-clocked at 240 MHz, compared to the 120 MHz maximum over-clocked speed of the Teensy 3.2. **Figure 1** shows the Teensy 3.6 module. **Figure 2** shows the Audio Shield that I used. It contains the NXP SGTL5000 Codec device (A/D and D/A converters, mic preamplification, headphone driver and digital signal processing).

CONVOLUTION FILTERING

Although I have used digital filters in FIR and Biquad configurations, prior to this project I wasn’t familiar with the term

“convolution” filtering. As part of my music/recording hobby, I had encountered the term convolution regarding:

- 1) Guitar amplifier cabinet simulation
- 2) High-end, “space-accurate” reverberation processors

Convolution reverberation processors are not relevant to this discussion. However, guitar amplifier cabinet simulation is basically a fancy way of saying that you are simulating the exact frequency/phase response of a guitar amplifier and its loudspeaker(s), mounted in a specific cabinet, with the recording microphone oriented a specific way.

The “shape” of the frequency response curve of any given guitar amplifier/speaker combination will not be a “flat” response over the useful range of guitar notes. Instead it will consist of many small peaks and dips over the frequency range of interest. These “aberrations” provide the distinctive sound of interest to the musician. To some extent, one can simulate a given guitar amplifier/speaker by using a multiband parametric equalizer (EQ) and fiddling with it until it sounds the way you know the actual amplifier/speaker sounds. However, experts in the field learned that they could go one step further using the following method.

Rather than feeding an actual guitar signal into the amplifier/speaker cabinet, they feed it a short pulse, with rise/fall times as fast as possible. This short pulse is called a “finite-impulse signal.” The sound emitted by the speaker cabinet is then picked up by a professional-quality microphone, amplified, converted to digital form and stored in a

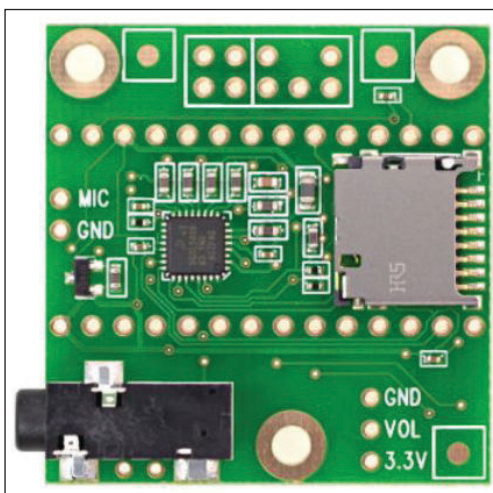


FIGURE 2

Top view of the Teensy Audio Shield. The two rows of 14 holes are fitted with header pins that plug directly into the Teensy 3.6 MCU module. All interconnections between the two boards are via these 28 pins.

file. This file represents the FIR of the guitar amplifier/speaker cabinet. I admit that I don't have the best understanding of the mathematical "magic" involved here, but suffice it to say that all the frequency response "personality" of the guitar amplifier/speaker cabinet is contained in the finite-impulse-response (FIR) file that has been collected. The higher the sample rate used to record the impulse, the better the simulation, and the larger this FIR file will be.

Once you have this FIR file, you can use it to provide the coefficients needed for a digital FIR filter. If you pass your "raw" guitar signal through this FIR filter, it will be modified in virtually the same way that it would be if it were sent out to the specifically modeled guitar amplifier/speaker cabinet. Effectively, you can digitally record a "raw" guitar signal, which, when converted back to analog and listened to it "live," through the specific guitar amplifier/speaker that you have modeled. The FIR filter routine does what's called a "convolution" of the guitar's time-domain signal with the FIR array of coefficients—which is also time-domain data.

FOCUSING ON FIR

Once you absorb the idea behind this simulation technique, it becomes clear that you could implement a complex digital filter to reproduce almost any complex frequency response with this technique. I'm certain that mathematicians and electronics engineers in the communication field discovered and used this technique to design complex filters long before guitar players saw its usefulness. However, it was the guitar cabinet simulation concept that led me to investigate the FIR filtering technique more fully.

It turns out that implementing a FIR filter with enough "taps" or coefficients to perform realistic guitar amplifier/cabinet simulation generally requires a FIR filter with 512 taps or more. The Teensy 3.6, running at 240 MHz (overclocked)—and using its built-in DSP 16-bit fixed-point instructions—can process a 100-tap FIR filter (using the Teensy Audio library's FIR filter block), using only 7% of available MCU time. This is for 16-bit data at a 44,100 Hz sample rate. That 7% figure is strongly influenced by the fact that most of the SGL5000 Codec data transfers (in and out) are done under DMA, which frees up the main MCU from performing this time-consuming task.

Because FIR filter's execution time is directly proportional to the number of taps [1], a 512-tap FIR should require 36% of available execution time. This timing seems reasonable, but implementing a FIR filter with such a large number of taps is impractical when using 16-bit fixed-point numbers. The accuracy is not nearly good enough to achieve proper results.

What is needed is a way to implement a floating-point 512-tap convolution process that is fast enough to handle 16-bit signals at a 44,100-Hz sample rate, in real time. A powerful set of math/DSP routines for Arm Cortex devices is contained within the Cortex Microcontroller Software Interface Standard (CMSIS) library. I made use of several floating-point math functions contained in the CMSIS-DSP library.

TIME VS. FREQUENCY

The previous discussion involved processing signals in the time domain. That is, we sample a signal at a fixed sample rate, process these data and then send the data out at the same sample rate. We could also do the electronic filter processing in the frequency domain. This would involve converting our time-domain signal into the frequency domain. This means doing basically the same filtering (but in a different way), converting the frequency-domain signal back into the time domain, and then sending it out. On the

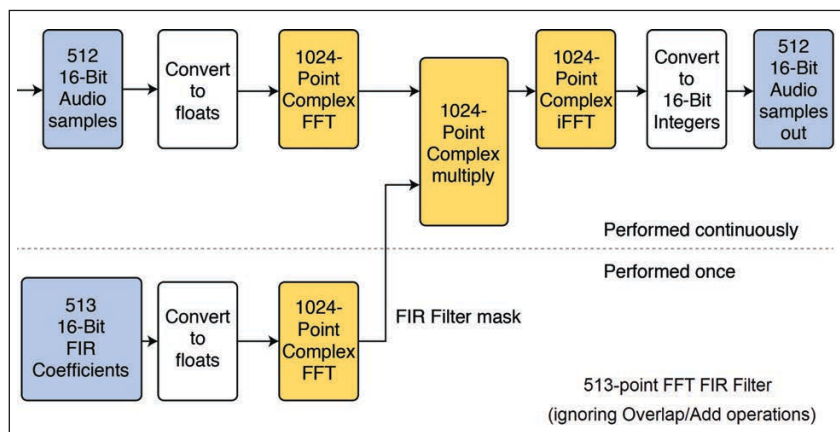


FIGURE 3

Block diagram of the algorithm used in the Convolution Filter. The details of the overlap-add operations are not shown here, but are explained in the article.

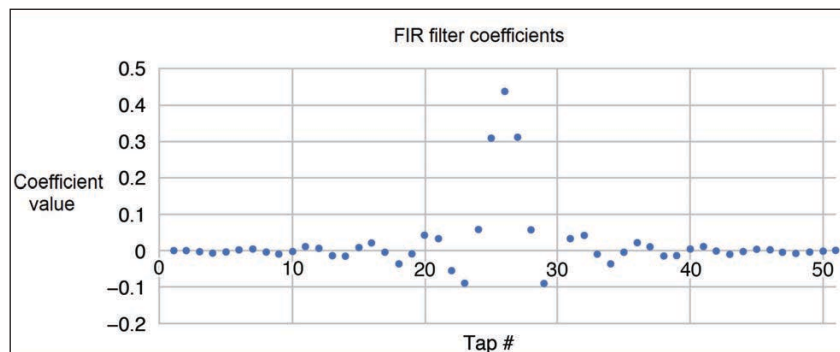


FIGURE 4

Shown here is a representative plot of the coefficients of a FIR low-pass filter. Notice that it is symmetrical around the half-way point in the number of taps.

surface, it would seem that this unnecessarily complicates the procedure, but there is a good reason to do it this way.

Converting the time-domain signal into the frequency domain can be done with a Fast Fourier Transform (FFT) routine. Converting it back into the time domain can be done with an Inverse Fast Fourier Transform routine (iFFT). Both the FFT and iFFT routines are available in the CMSIS DSP library available for Arm MCUs. For the Cortex M4F cores with built-in floating-point operations, the applicable CMSIS libraries perform those operations in floating point, very efficiently.

The big advantage to doing the filtering in the frequency domain rather than the time domain is that the computationally intensive convolution routine can be replaced by a matrix multiply routine. I referenced Steven Smith's *The Scientist and Engineer's Guide to Digital Signal Processing* [1] while doing this project. A link to it is available on the Circuit Cellar article materials webpage. In Chapter 18 he mentions that the execution time for a standard FIR convolution routine is proportional to the number of FIR "taps," whereas an FFT convolution routine's execution time increases only as the logarithm of the number of FIR taps. Smith assumes that equivalent floating-point math instructions are used for both methods, and the following holds true:

- 1) For < 64 taps, standard convolution routines are faster.
- 2) For > 64 taps, FFT convolution routines are faster.

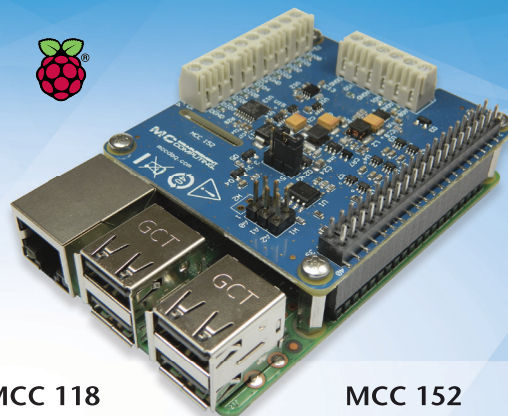
In figure #18-3 of Smith's text [1], he shows that a 512-tap standard convolution is almost 4 times slower than a 512-tap FFT convolution. I had no idea how much slower the Teensy 3.6's floating point instructions would be compared to its highly-optimized DSP 16-bit fixed-point instructions. Therefore, I couldn't tell whether it would be possible to implement a standard 512-tap floating-point FIR filter in real time (at 16-bit, 44,100 Hz sample rate). Considering that the FFT convolution routine should be 4 times faster, I decided to use that technique. Looking at the result that I show later in the article, this proved to be a wise choice.

BASIC IMPLEMENTATION

Figure 3 shows the basic algorithm used for a 513-tap FFT FIR convolution filter. First let's consider the 513-tap figure. When doing a convolution, the filter "kernel" that is used must be symmetrical around its central point (**Figure 4**). That is why a 513-tap value (an odd number) is used rather than 512. 512 is 2^9 (FFTs are generally 2^n in size).

Before doing any processing on the audio input stream, we must first obtain a "filter mask." This is derived from the array containing the FIR filter coefficients—after it has been processed by a floating-point complex FFT routine, which brings it into the frequency domain. In Figure 3, I show the 513-point FIR coefficients as a 16-bit integer array. That is how a guitar cabinet impulse response file is structured—it is supplied as a WAV file in 16-bit signed format. I convert this to a floating-point array (using CMSIS `arm_q15_to_float`), so that it can be processed by the 1024-point, floating-point complex FFT routine (CMSIS `arm_cfft_f32`). Note that if you were instead trying to implement a FIR filter using coefficients from a FIR filter calculator [2], they would be normalized floating-point numbers. My FIR Filter Mask processing routine expects 16-bit integer values, so you would have to multiply those normalized floating-point coefficients by 32,768. The FIR Filter Mask, as described above, needs to be calculated only once for any given FIR filter

Bring Your Pi to Work!



MCC 118 Analog Input

- 8 channels
- 100 kS/s
- 12-bit resolution

MCC 152 Analog Output

- 2 channels
- 8 DIO

Measurement Computing now offers DAQ HATs for Raspberry Pi®

MCC Raspberry Pi measurement HATs were designed to bring high-quality measurements to the ubiquitous low-cost computer. MCC DAQ HATs are the perfect solution for adding professional quality DAQ capabilities to the Pi platform.

MCC DAQ HATs include:

- Complete SW library for easy programming
- Full set of examples in C® and Python™
- Stackable for high-channel count
- Quality from a trusted source
- **Perfect for:** embedded system design, industrial IoT, end-of-line test, and more...

www.mccdaq.com/DAQ-HAT



MEASUREMENT COMPUTING™

profile. You might wonder why I am using a 1024-point complex FFT routine, when I have only 513 data-points. I'll discuss that later.

Next, let's look at the processing needed for filtration of the signal in real time. The Teensy Audio library does all its audio data transfer and processing in 128 blocks of 16-bit audio data. This means the incoming digital audio signal (from the SGTL5000 Codec) is transferred into Teensy 3.6 SRAM by a DMA burst transaction of 128 words (256 bytes). Similarly, these 128-word blocks are moved between various SRAM memory locations under DMA control for processing. Finally, the output data also are sent back to the Codec under DMA control.

This block size is a compromise chosen to minimize latency time (2.9 ms per 128-sample block,) while still allowing for efficient DMA transfers and other data-processing chores. However, for the 513-tap FIR routine to work, we need our 16-bit audio data to be available in 512-sample blocks. Without going into any detail yet, let's just say that four of the Teensy Audio library's 128-sample blocks are concatenated into one 512-sample block. An integer-to-float routine (`CMSIS_arm_q15_to_float`) is used to convert this into a 512-element floating-point array.

This 512-sample array of time-domain audio data must now be converted into the frequency domain. This is done using a 1024-point complex floating-point FFT (`CMSIS_arm_cfft_f32`). Why do we need a 1024-point complex FFT when we are processing only a 512-sample audio block? To begin, the audio signal data coming in consists of only the real part, not the imaginary part of a *complex* array. The math behind this is beyond my pay grade. But I know from experience that the sound coming out of the filter won't be correct if you don't use a *complex* FFT routine, and you must fill the imaginary portion of the input array with the same audio data that you have in the real portion. The complex FFT routine expects its input array to have the real and imaginary values interweaved, so when you are transferring the incoming audio data into the FFT array, you write each value twice before advancing to the next incoming data point.

The second question here is why are we doing a 1024-point FFT on only 512 input samples? Where are we getting the extra 512-points that we need to present to the 1024-point FFT? Here again, the theory is somewhat above my pay grade, but this is how I understand it.

BACK TO TIME DOMAIN

Let's go back to thinking in terms of a time-domain signal. If we are considering a continuous stream of digital audio data, it

is obvious that the MCU cannot process the continuous data stream all at once. We must break the signal into smaller blocks and do the filtering on each block individually. Without getting into any math, I think it's intuitive that filtering is just doing some form of weighted averaging over several data-points. At the very start of the datastream, there won't be any "past history," so the averaging process won't be accurate. But that only happens once, at start of processing. The middle section of the block will filter okay, but as we get toward the end of the block, we'll be missing the data present at the start of the next block, so that the averaging (filtering) will again be inaccurate. We therefore need to process the data in a way that takes into consideration the data from the next 512-sample block of data.

When a FIR digital filter with a 100-point filter kernel processes 100 incoming data points, it will result in an output of 200 data-points. Obviously, we can't send out 200 data-points for every 100 data-points coming in, given that the input and output sample rates are identical. If you analyze the math involved, it turns out that to provide an accurate filtered signal you must:

- 1) Break the incoming signal into a block half the size of the FIR filter kernel.
- 2) Add a block of zeros to the end of these signal data, to make the total length equal to the size of the filter kernel.
- 3) Perform the FIR filtering on this block, resulting in an output block equal to twice the size of the filter kernel.
- 4) Send the first half of this output block out to the Codec, and save the last half of this block for later.
- 5) Perform steps 1, 2 and 3 again on the next incoming block of data. However, for step 4, recover the saved block of data from before, add it to the first half of the output block, then send this composite first half block out to Codec. Save the second half of the block for later (as in 4).

This process is referred to as the overlap-add method in DSP texts.

When we consider the FIR convolution process being done in the frequency domain, similar considerations will apply. We take 512 samples of the audio data and place it in the first half of the 1024-point FFT input array, filling both the real and imaginary elements with the audio data as mentioned above. We then fill the second half of the array with zeros (for both the real and imaginary elements). After the 1024-point FFT is performed, we will have a 1024-element of complex data in the frequency domain. In a similar fashion,

the 513 FIR coefficients are padded out to 1024-points before undergoing the 1024-point FFT—which produces the Filter Mark.

The FIR convolution process in the time domain is equal to an array multiplication in the frequency domain. So, we take the FFT array from the incoming signal and multiply it with the FFT array from the FIR filter coefficients (the Filter Mark that were pre-calculated). The resulting 1024-point array, still in the frequency domain, must now be converted back into the time domain. This is done using a 1024-point iFFT routine (CMSIS `arm_cfft_f32`). Note that both the CMSIS FFT and iFFT routines are called using the same “`arm_cfft_f32`” label, but there is a

parameter passed to this routine for which a “0” designates an FFT and a “1” designates an iFFT routine.

We are now back in the time domain with an array of 1024 floating-point digital audio samples. We take the first half of this array and add it to the 512 points of data saved from the last block. These 512 floating-point numbers are then converted back to 16-bit integers (CMSIS `arm_float_to_q15`) and sent out to the Codec to be converted to an analog signal. We then save the second half of this array to a temporary array, which will be added into the output stream the next time around. You can see that the overlap-add method that I discussed in terms of the time-domain FIR convolution is also performed, in

```
// 4 blocks are in- now do the FFT1024,complex multiply and iFFT1024 on 512samples of data
// using the overlap/add method
// 1st convert Q15 samples to float
arm_q15_to_float(buffer, float_buffer_L, 512);
// float_buffer samples are now standardized from > -1.0 to < 1.0
if (passThru ==0) {
    memset(FFT_buffer + 1024, 0, sizeof(FFT_buffer) / 2);
// zero pad last half of array- necessary to prevent aliasing in FFT
//fill FFT_buffer with current audio samples
k = 0;
for (i = 0; i < 512; i++)
    {
        FFT_buffer[k++] = float_buffer_L[i]; // real
        FFT_buffer[k++] = float_buffer_L[i]; // imag
    }
// calculations are performed in-place in FFT routines
arm_cfft_f32(&arm_cfft_sR_f32_len1024, FFT_buffer, 0, 1); // perform complex FFT
arm_cmplx_mult_cmplx_f32(FFT_buffer, FIR_filter_mask, iFFT_buffer, FFT_length);
// complex multiplication in Freq domain = convolution in time domain
arm_cfft_f32(&arm_cfft_sR_f32_len1024, iFFT_buffer, 1, 1); // perform complex inverse FFT
k = 0;
l = 1024;
for (int i = 0; i < 512; i++) {
    float_buffer_L[i] = last_sample_buffer_L[i] + iFFT_buffer[k++];
        // this performs the “ADD” in overlap/Add
last_sample_buffer_L[i] = iFFT_buffer[l++];
//this saves 512 samples (overlap) for next time around
    k++;
    l++;
}
} //end if passThru
// convert floats to Q15 and save in temporary array tbuffer
arm_float_to_q15(&float_buffer_L[0], &tbuffer[0], BUFFER_SIZE*4);
```

LISTING 1

Most of the actual computation is performed in this section of the program. The complexity is hidden by the use of high-level, DSP-like routines contained in the Arm CMSIS library.

a similar way, in the frequency-domain FIR convolution process. Note that in Figure 3, I've simplified the diagram somewhat by not including the zeroing of the second of the signal input array (and Filter Mask routine) nor have I shown the addition of the saved arrays from the previous block calculations. **Listing 1** shows the "C" program code to perform the filtering as explained above.

IMPLEMENTATION DETAILS

The above description assumes that 512 audio samples are available to filter, all at once. However, the Teensy Audio library doesn't work this way. It operates with a timed interrupt service routine (ISR) that occurs every 2.9 ms and processes a single, 128-sample block of audio data.

All the Teensy Audio processing libraries must contain a routine called "update." This routine is responsible for receiving one of these blocks, doing whatever processing is required, and then transmitting that block and releasing its memory. You can use numerous

Audio library functions in series, if so desired. So, every 2.9 ms, the Audio ISR fires, and the update code for each of the audio functions that the programmer has used in the program will be executed in sequence. Each one is processing a single, 128-sample block of audio data, and then passing it along.

Obviously, I had to write some code to adapt this 128-sample block processing into one that works with 512 samples at a time. To do this, I define a variable called "state," which persists between these Audio ISR "update" calls. At each update, "state" is incremented by 1. For states 0 to 3, I store the incoming 128-samples of audio data in a temporary 512-element integer buffer (incrementing the buffer pointer by 256 bytes each time).

When state=3, this temporary buffer is full, so I call the 512-point FFT convolution routine (described in the last section and shown in Listing 1). That fills up a 512-element integer transmit buffer. The state variable is now set to zero, to start the process over again. In addition, for states 0 through 3, I point to

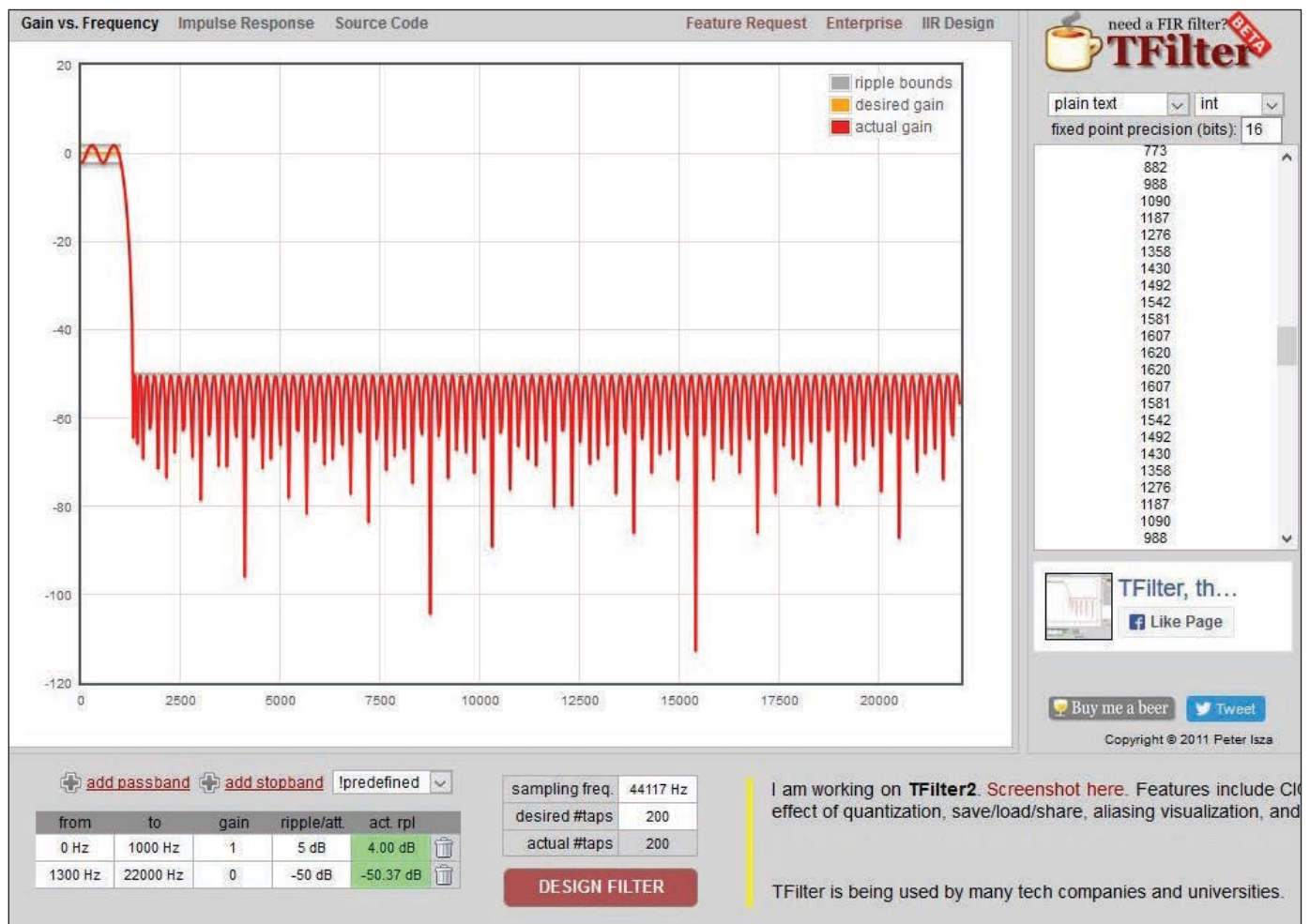


FIGURE 5

A screen-capture of the Web-based program TFilter. This program can be used to generate FIR filter coefficients for various types of digital filters. To the right, you can see I've selected integer coefficients, because that is what my program expects. But floating-point numbers can also be chosen.

successive one-quarter sections of this transmit buffer, and send a 128-sample block from this buffer section back out to the Audio library's queue, where it will either undergo further processing (if required by the program) or be sent out to the Codec to be converted to an analog audio signal. The transmit buffer will have no valid data in it the first four times that the Audio update occurs, since no filter processing has yet taken place. So, you could get a short "blip" of noise (around 12 ms) when the program first starts processing audio data.

If you've carefully followed the above explanation, you can see that out of four consecutive, ISR-driven "updates," three of them do no processing apart from moving data from one buffer to another. It is the fourth update that does all the filter processing. Using the Audio library's `AudioProcessorUsage()` function, I found that the percentage of available MCU processing power used by updates 1 through 3 was less than 1%, and update 4 was 47%. These figures are obtained with the Teensy 3.6 overclocked at 240 MHz. The figures—quoted on my original GitHub site for this project [3]—are for a Teensy 3.6 clocked at 180 MHz, and are proportionately higher.

GENERATING THE FILTER MASK

Earlier, I explained that the desired FIR coefficients must be converted into what's called the Filter Mask, for frequency-domain filtering. Basically, I was interested in two sources for these FIR coefficients:

- 1) FIR filter coefficients for standard types of filters, obtained by filter calculation programs—either web-based tools or dedicated programs running on either a PC or an embedded MCU
- 2) Guitar Cabinet Impulse files

Let's look at #1 first, because this type of filtering could be used more widely. If you need a filter with specific parameters that will seldom or never change, you are probably best served using a FIR filter design application, either web-based or a PC application that can be downloaded. A common web-based program is TFilter [2].

Using this program, there are a few considerations to note. For use with the Teensy Audio library/Audio Shield, the sample frequency must be set to 44,117 Hz. The Teensy Audio library actually runs at a sample rate of 44,117 Hz, slightly different from the CD standard of 44,100 Hz. Also, the filter coefficients will be output in either double-precision floating-point or integer. You would choose integer in this case, as my program is designed to work primarily with Guitar Cabinet Impulse files, which are normally formatted as Microsoft WAV files. These files use a 16-bit waveform format. **Figure 5** is a screenshot of TFilter showing a low-pass filter.

If the parameters of the filter must be changeable while the Teensy 3.6/Audio Shield is running, then another approach must be taken. If you needed only a few FIR filter profiles, it would be possible to pre-calculate them using TFilter, and then load several banks of FIR coefficients into flash memory, to be switched in and out of SRAM as needed. The Teensy 3.6 contains 1 MB of flash memory, so there's plenty of room for filter coefficient banks.

Another approach is to embed a FIR filter calculation routine in the Teensy 3.6's code, itself. I have included a Teensy program that includes the `calc_FIR_coeffs` function. This routine calculates floating-point FIR coefficients for Low-pass, High-Pass and Band-pass filters, for a user-selected number of FIR taps. Since this routine provides normalized floating-point coefficients, I multiply all the values by 32,768 before sending them to the "cabinet_impulse" array (a 16-bit integer array).

The parameters passed to this routine are as follows:

```
calc_FIR_coeffs (float * coeffs, int numCoeffs, float32_t fc,
float32_t Astop, int type, float dfc, float Fsample)
```

`float * coeffs`: a pointer to a `float32_t` array large enough to handle the designated number of coefficients (taps)

`int numCoeffs`: an integer specifying the number of coefficients

`float32_t fc`: a floating-point number specifying center or cutoff frequency

`float32_t Astop`: a floating-point number specifying expected stopband attenuation in dB

`int type`: type of filter- 0-Low-Pass 1-High Pass 2-BandPass

`float dfc`: a floating-point number specifying half-filter bandwidth (for BandPass only)

`float Fsample`: a floating-point number specifying the sample rate in Hz.

For Cabinet Impulse FIR coefficients, the coefficients are generally stored in a Microsoft WAV file. The ones I have seen contain enough data to fill a 513-tap FIR filter coefficient array. For some reason, the ones I have seen are often very long files—many hundreds of thousands of bytes or more. Of this, only the first 512 or so data points in the “wave” chunk of the file contain actual coefficient data. The rest are zero-padded. Microsoft WAV files do not just contain raw wave data—they also include a header section at the beginning of the file. This header section contains “meta-data” about the format of the file, a pointer to the start of the wave data, and the length of the wave.

For my Teensy 3.6 application, I place the WAV file containing the Cabinet Impulse file onto an SD card. This card must be inserted into the SD CARD socket on the Teensy 3.6, itself—*not in the SD card socket found on the Audio Shield*. In the program, I open the file “MG.WAV,” which is the name of the sample file I used. You must modify this line of my program to match the filename you have, or rename your file to match.

To find the start of the wave data, I open the file and search for the string “data.” Assuming it is a true WAV file, the string “data” should be found. I then skip over the next 4 bytes (the wave data size field) and then read in 513 integer values. These are stored in the array `cabinet_impulse` (type `int16_t`). Whichever method you use to generate the FIR coefficients, the coefficient data in the `cabinet_impulse` array must be converted into a frequency-domain Filter Mask. This is done, in the Setup portion of the program as follows:

```
convolution.begin(0);
// set to zero to disable audio processing until impulse has been loaded
convolution.impulse(cabinet_impulse);
// generates Filter Mask and enables the audio stream
```

Once `convolution.impulse` has executed, a valid Filter Mask array will exist, and the real-time processing (filtering) of the incoming audio stream will begin. **Listing 2** shows the

```
void AudioFilterConvolution::impulse(int16_t *coefs) {
  arm_q15_to_float(coefs, FIR_coef, 513); // convert int_buffer to float 32bit
  int k = 0;
  int i = 0;
  enabled = 0; // shut off audio stream while impulse is loading
  for (i = 0; i < (FFT_length / 2) + 1; i++)
  {
    FIR_filter_mask[k++] = FIR_coef[i];
    FIR_filter_mask[k++] = 0;
  }

  for (i = FFT_length + 1; i < FFT_length * 2; i++)
  {
    FIR_filter_mask[i] = 0.0;
  }
  arm_cfft_f32( &arm_cfft_sR_f32_len1024, FIR_filter_mask, 0, 1);
  for (int i = 0; i < 1024; i++) {
    // Serial.println(FIR_filter_mask[i] * 32768);
  }
  // for 1st time thru, zero out the last sample buffer to 0
  memset(last_sample_buffer_L, 0, sizeof(last_sample_buffer_L));
  state = 0;
  enabled = 1; //enable audio stream again
}
```

LISTING 2

The `convolution.impulse` routine takes a 513-element FIR array (integer) and converts it into a 1024-element Filter Mask (floating-point). The CMSIS complex FFT routine is used for this purpose.

convolution.impulse routine. The routine is passed a pointer to the 513 element FIR coefficient array generated as described above. It first converts this integer array to a floating-point array. Then it fills up the first 513 real elements of the 1024 element FIR_filter_mask array with those 513 coefficients. Since the FIR_filter_mask array must hold complex values, every second element is set to zero—in other words, zeroing out the imaginary part. The final 511 complex elements of this array are also zeroed out. The rationale for zeroing out of the last part of the array is explained in Smith's text [1]. The filter produces a lot of artifacts in the signal if this is not done!

After the 1024 element array has been prepared as above, a complex FFT is performed on the array (CMSIS arm_cfft_f32). Part of the “magic” in these CMSIS FFT routines is that they do the FFT process “in place”—in other words, no separate array is needed for the transformed result. As a last step, this routine zeroes out the last_sample_buffer array, which is used in the overlap-add process mentioned earlier. The first time through the overlap-add process, there is no valid last_sample_buffer array data, so it needs to be zeroed out.

PROGRAM DETAILS

A few program details merit discussion. The FFT convolution filter that I wrote is structured to work with the Teensy Audio library. When the Teensyduino Arduino add-in is installed, this Audio library will be installed by default—unless you specifically un-check the box corresponding to it during the installation routine.

Two things must be done to the Audio library to include this convolution filter:

- 1) You must install some CMSIS files to the Teensy core library. The exact procedure for doing this can be found in the text file “Adding CMSIS 4 library files” located at *Circuit Cellar's* article code & files download webpage. Alternately, instructions can be found on my GitHub site [3]. I also include alternate instructions to incorporate the newest CMSIS 5.3 library. Either one will work properly.
- 2) The convolution filter code consists of 2 files: `filter_convolution.h` and `filter_convolution.cpp`

These files must be added to the folder containing the Teensy Audio library. This folder will be located under whatever folder you have installed the Arduino/Teensyduino IDE. The path is: `c:\your_arduino_folder\hardware\teensy\avr\libraries\Audio`

Also, in that folder, edit `Audio.h` by adding the following line at the end:

```
#include "filter_convolution.h"
// library file added by Brian
// Millier
```

Like any custom Audio library objects that you add yourself, this one will not show

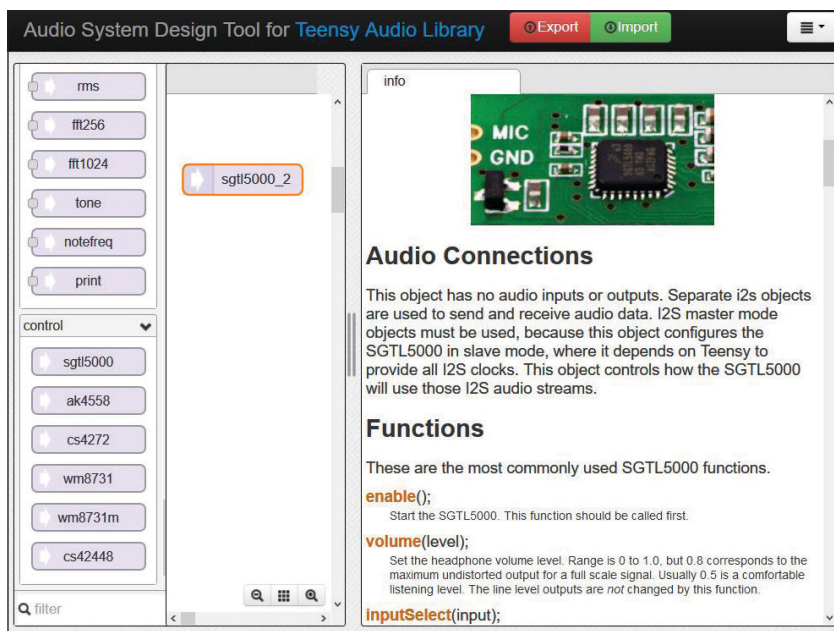


FIGURE 6

An easy way to familiarize yourself with the SGT5000 Codec used in the project is to refer to its Help file in PJRC's online Audio System Design Tool.

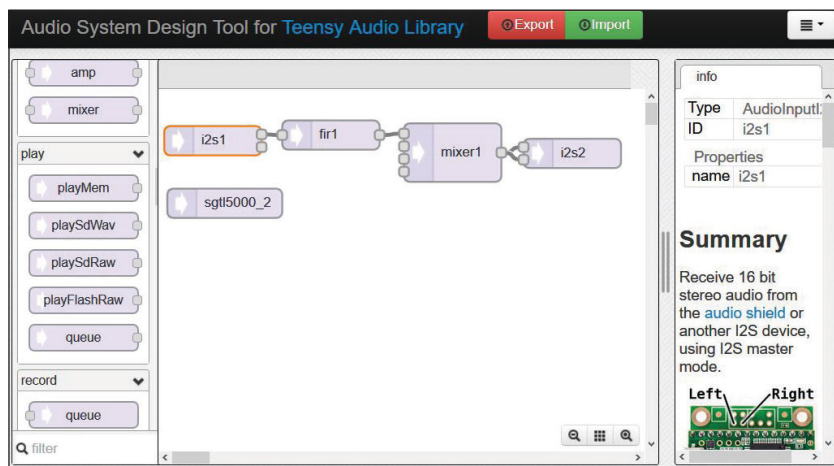


FIGURE 7

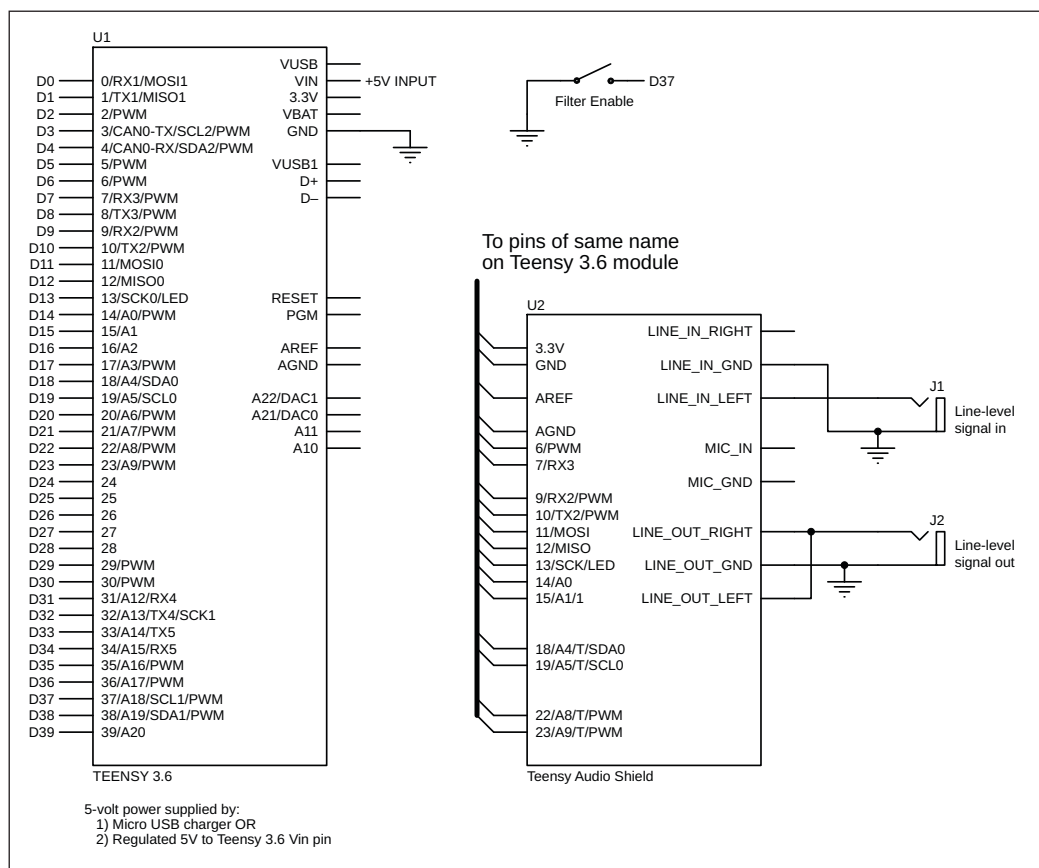
The Audio System Design Tool's workspace for this project. Note that the FIR object is shown. See article text for explanation.

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.

FIGURE 8

Schematic diagram of the hardware used for this project. It consists of only a Teensy 3.6 module with a PJRC Teensy Audio Shield mounted on it.



up in the Audio System Design Tool found on the PJRC site. Probably the easiest way to generate the setup/connection code needed to incorporate this filter into your audio configuration, is to draw your configuration using the Design Tool Web program, but place a FIR filter. Import this configuration into your sketch. Then, within the “// GUItool: begin automatically generated code” replace “AudioFilterFIR fir1” with “AudioFilterConvolution convolution”. Also, on two of the AudioConnection lines, replace instances of “fir1” with “convolution”

In my sample program, this configuration has already been done. The above procedure is only needed if you are writing your own program using additional Audio library objects. If you want the convolution filter keywords to be highlighted in orange in the Arduino IDE (like all the other Audio library objects), you

can add the following line to the keywords file (contained in the Audio folder):

```
AudioFilterConvolution<TAB>  
KEYWORD2.
```

Note that you must separate these two words with a TAB character, not with spaces.

THE SGTL5000 CODEC

I'll just mention a few details about the NXP SGTL5000 Codec found on the Teensy Audio Shield. It contains both Line input and Microphone inputs. The Microphone input is configured for an electret microphone (DC bias is provided). The SGTL5000 has a programmable gain preamplifier for the Microphone input. Both Line out and Headphone outputs (stereo) are available, and the Headphone output channel has a wide-range volume control, which is adjusted under program control. The SGTL5000 contains its own Digital Audio Processor (DAP)—basically a specialized DSP that can perform various EQ and Auto Level Control functions. An easy way to become familiar with the capabilities/settings for this device, is to access the online program Teensy Audio Library Design Tool. See the *Circuit Cellar* article materials webpage for the link.

When using the Audio Shield, your sketch must contain the SGTL5000 control object. When

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

References [1] through [3] as marked in the article can be found there.

RESOURCES

NXP Semiconductors | www.nxp.com

PJRC | www.pjrc.com

this is included, all the necessary initialization code will be added to set up the SGTL5000 in a default configuration. The SGTL5000 is configured via the I²C bus. Its I²C address is 0x0A, which shouldn't conflict with most other I²C devices that you might also want to use.


The easiest way to learn about the SGTL5000's capabilities and programming is to use the Teensy Audio Library Design Tool. **Figure 6** is a screenshot of the Audio Library Design Tool, showing a bit of the SGTL5000 info screen on the right. **Figure 7** is a screenshot of the Audio Library Design Tool configured for this project. Note that a standard `fir1` filter object is placed in the workspace. See the explanation in the prior section on how to replace the code generated by the `fir1` object, with code that implements the Convolution filter instead.

Figure 8 is the schematic of the project. As you can see, it comprises two modules: A Teensy 3.6 MCU module and the Teensy Audio Shield. The Audio shield is designed so that it can be mounted on the Teensy 3.2, 3.5 or 3.6 MCU modules directly—eliminating any interconnecting wiring. The audio Line In and Line Out are available on a 10-pin IDC header. The signal designations are shown on the bottom of the board, and the pinout

matches that of a PC motherboard's audio Line in/out connector. A filter In/Out switch is connected to the Teensy Digital 37 GPIO pin. A 5-V power source can be applied either via the micro-USB port, or the V_{in} pin on the Teensy 3.6 module.

CONCLUSIONS

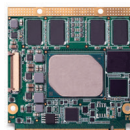
I believe I spent more time figuring out how to write this code than on any other non-work-related program I've tackled. The final code seems very simple, because it makes extensive use of the CMSIS library routines. However, learning how they worked and how to integrate them into the pre-existing Teensy Audio Library was quite challenging. On the other side of the coin, building the circuit was trivial due to the easy integration of the Teensy 3.6 MCU module with the PJRC Audio Shield.

Author's Note: I'd like to acknowledge all the programming effort of Paul Stoffregen, who wrote the Teensyduino Arduino add-in and the core of the Audio Library. I also referenced work done by Frank (DD4WH) on his Teensy SDR project, which included similar FFT convolution routines. A link to Frank's Teensy SDR project can be found on the Circuit Cellar article materials webpage. 

www.congatec.com/us
info@congatec.com
Phone: 858-457-2600



Fast Play - Jackpot!



conga-QA5

New power saving Qseven computing module.

Fun for life.

- Latest Generation Intel® "Apollo Lake" CPUs
- High performance 4k@60Hz graphics & H.264 encoding
- Lowest power consumption for passive cooling
- Personal integration support included

Embedded in your success.

From the Bench

An Itty Bitty Education STEM at Home

There's no doubt that we're living in a golden age when it comes to easily available and affordable development kits for fun and education. With that in mind, Jeff shares his experiences programming and playing with the Itty Bitty Buggy from Microduino. Using the product, you can combine Lego-compatible building blocks into mobile robots controlled via Bluetooth with your smartphone.

By
Jeff Bachiochi



FIGURE 1

The Itty Bitty Buggy phone app provides both “build”: step-by-step assembly of each creature and “play”: the Bluetooth controls to make your creature come alive.

I don't know a kid who hasn't played with Legos. I even know adults who still enjoy building something with Legos. I know because I'm one of them. When Christmas comes around, I always try to incorporate my love of Legos and fascination with robots into gifts for my grandkids. I wish I had the bucks to give every one of them Mindstorm EV3s. But at \$350 each (times 13) there's just no way I can swing that and eat too. So, I keep my eyes peeled for things that are a little less expensive but still seem to have a good value to them. This year I found the Itty Bitty Buggy (IBB).

This is a product put out by Microduino. You may know that Microduino makes a similar product that's about one-fourth the size of an Arduino Uno yet has a similar bus structure. In other words, you can stack expansion boards atop the core microcontroller (MCU) board. There are more than 100 modules, sensors and actuators to solve most application requirements. As you might have guessed, you can use the Arduino IDE to program the Microduino series of parts. In addition, they offer mDesigner, which is Scratch-based and designed for young minds with no programming experience.

The IBB is Microduino's newest product. They've taken the best of what they've learned and incorporated it into an inexpensive learning tool. Listed at \$59, it fits my requirements for immediate fun, with a path for educational growth. First, we'll take a quick look at the fun part.

CELL PHONE = FUN

The Itty Bitty Buggy is a mobile vehicle controlled via Bluetooth using your Android or iOS device. The app provides instructions for building and playing with the Buggy in one of five scenarios: Buggy movement (wheels), Dodo Bird (flapping wings), Sloth (suspended rope walker), Ladybug (legs) and Alien (arms). There are on-screen instructions for the simple “build” of each of the five critters. Each build instructs the user on how to assemble each character, in easy-to-follow pictorial steps. It has four modes: remote control (joy stick), line following (color sensors), voice control (phone microphone) and music (joy stick and color sensors). The other four scenarios are based on the buggy scenario—the difference being that each creature has different movements, based on similar controls.

No programming knowledge is needed for any of these scenarios. The hardest part is pairing your phone to the Buggy's Bluetooth module! Once you've done this, the Buggy will make contact with your phone and is ready to perform its control of any of the preprogrammed scenarios.

If you've looked at the app's screen in **Figure 1**, you may have noticed the word “program” at the top right of the screen. This button brings up the next level of control available to a user. So, after you had enough fun playing with all the scenarios prewritten on the IBB, hit “program” and let's dive deeper.

Let's assume we have the simple buggy built and it

only has wheels. You can start a new project by tapping “+.” But instead tap “project1”, which has already been created for you (**Figure 2**). In programming mode, you create or alter projects using drag-and-drop selections from one of the menu icons: Switch (procedures), Control (repeat), Calculation (comparisons), Light (colors), Sound (beeps), Movement (drive) and Sensor (colors). Each project is a program made up of blocks placed on the project field. On the field’s left are two buttons: Stop (yellow square inside a red button) and Start (yellow right pointing triangle inside a blue button). On the field’s right are menu icons.

Each project should begin with the Start block. This has an icon of the start button on it and is found in the procedure group displayed when the switch icon is touched. Project1 has the Start block already on the project field, along with a few other blocks. Just below the Start block you’ll find two Light blocks and one Sound block. If you look at the Light blocks, you’ll see each one uses a different sensor: A (the driver-side LED/sensor) and B (the passenger-side LED/sensor). Each of these has an associated color. You can choose a different color by tapping the color swatch, tapping a new color from the pop-up menu and tapping the project field to select it. The Sound block allows you to select a note and to adjust its duration using similar taps.

To have the IBB execute this program, just tap the Start button in the lower left of the field. The blocks are executed from the top down, turning on the left LED, turning on the right LED and playing a note. You’ll see that when the note has finished playing, it sits there waiting to execute more blocks, but since we have run out of blocks to execute, the buggy waits. Tapping on the Stop button will halt the program—the LEDs are turned off.

You can alter this project to investigate all the menu groups, and when you leave (Exit),



FIGURE 2

Should boredom set in, switch to the programming mode. Here you will drag and drop instructions to make the creature do your bidding.

the project will be saved if it has been changed. There is plenty here for a kid to really begin to understand how programming is simply a list of instructions to follow. You will note that the Start button is grayed while the program is executing. Presently, the scrollable project field can’t be resized, which means that larger projects can’t be seen without scrolling. This makes them a bit difficult to see. When you need your phone back, you can move the learning from the phone app to a PC. We’ll explore those options right after looking more closely at the actual hardware.

BREAKING DOWN BUGGY

The Buggy is divided into two modules, which I call the head and body (**Figure 3**). The head contains a CPU with USB (wired) and Bluetooth (wireless) interfaces. All I/O is through two bus systems: the “mCookie” bus, which uses “pogo pins” and magnets to ensure connectivity between snap-together modules, and the sensor bus, which offers a number of 4-pin connectors carrying power/

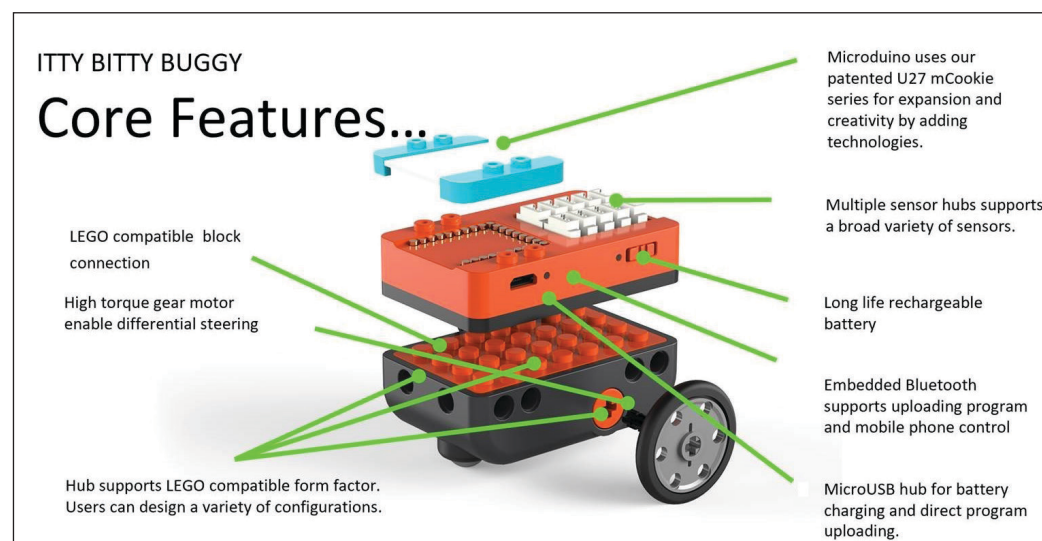


FIGURE 3

The Buggy has many built-in features. I like the lithium ion power source. You can add features by stacking a “cookie” or cabling a sensor via the top of the Buggy’s expansion buses.

FIGURE 4

Multiple cookies were combined to create a single PCB inside the top of the Buggy. The back of the PCB holds the two expansion buses. The front side holds most of the electronics (see inset).



ground and two analog or digital I/Os. And speaking of power, an integrated lithium battery—which charges from the USB port—provides long run times and so there's no worries about replacing batteries (**Figure 4**).

The mCookie is a modular, stackable, building-block-compatible electronics platform that supports Arduino. The original series consisted of a CPU “core” module and several stackable expansion modules such as communication, functions and sensors. Because several modules have been incorporated onto the Buggy’s “mCenter+” PCB—CPU, communication, battery management, sound and so on—the stackable expansion can be used to add module-like GPS, RTC or Lego NXT interfacing. A typical module—such as the 128 × 64 OLED—is priced at \$15.

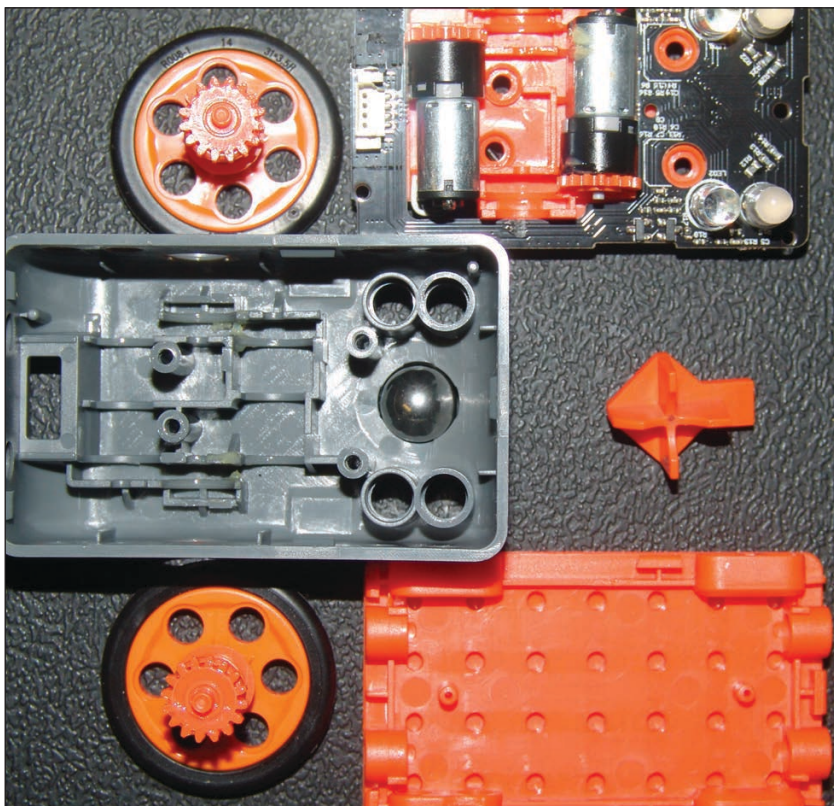
The sensor bus breaks out all the CPU’s I/O ports. The tiny connector is a 4-pin JST 1.25 mm. This connector is compatible with the Microduino line of sensors. You can choose from a long list of inexpensive switch, LED, environment and other input and output sensors. The only downside I see is that, unlike Lego sensors that are integrated into Lego bricks, these are tiny PCBs with exposed sensors. A typical sensor—such as the digital temperature sensor—costs \$4.

The body of the Buggy contains two motors, two LEDs and two color sensors. All this is multiplexed through one dedicated 4-pin I/O connector. Note in **Figure 5** that the wheel shafts are gear-driven 1:1 from the motors, and each axle has a mechanical clutch to prevent breakage. This connector could have been eliminated—or at least internalized—if the Buggy had been a one-piece unit. I’m not sure of the reasoning for breaking this into two pieces.

You may have noticed that you didn’t see any way for the phone app to handle any of these additional I/O thingies. And you’d be correct. For some kids, just beginning to learn about programming and playing with the available blocks on the phone app will be sufficient for a good while. In fact, not every child will develop the curiosity to come out and ask: “What now?” And that’s OK! That said, don’t toss out the Buggy just yet because there is plenty more for those interested in learning more. You can find additional information about the modules and sensors I have already mentioned on the Microduino website.

GRADUATION

Those who wish to graduate on to bigger and better adventures should grab their imaginations and download mDesigner from Microduino. After starting the application, you will find a screen divided into five areas: a toolbar (top-open/save/kit/port/settings),

**FIGURE 5**

The lower half of the Buggy contain motors and wheels, along with the color LEDs and sensors. The motor and color functions are controlled by one 4-pin cable (2-I/Os).

panes (left-code blocks/costume/sound), scripts (center-build area), stage (upper left-stop/start/sprite) and sprites (lower right-selecting sprites and backdrops). Realize that this environment was created to teach someone how to program by dragging and dropping instruction blocks onto the script area, not necessarily for the IBB. With that in mind, say you have a child that could not care less about robots. Invite them in right now. This is for them.

You can set aside the Buggy for now—it's not required for this exercise. Make sure it says "online" on the right side of the toolbar. If it says offline, just click it. I'll explain this later. Let's begin by creating a new project that any kid will have fun with. Click "Create a new Project" in the toolbar. Now click the Costume tab in panes. Click "Choose a Costume" at the bottom of panes. Scroll through the icons and pick "Dog2." This will be our Sprite. Next, click Stage on the right in sprites and the Costume tab will change to Backdrops. Click "Choose a Backdrop" at the bottom of panes. Scroll through the icons and pick Theater. Again click "Choose a Backdrop" at the bottom of panes. Scroll through the icons and pick Farm. Okay, now we can start coding.

Click the Code tab in panes. One the left of side of panes there is a column of functions that you can hit to scroll the associated instruction "blocks" into view in panes. These are similar to those in the phone app described earlier. Here I'll describe a selected block by using the syntax "function:block" to help you find it. Then place it in the script area and edit items in parentheses as necessary.

```
Events:when clicked
Looks:Say 'You click the mouse and I'll follow.'
Control:wait until <>
(Please Sensing:mouse down? into <>)
Looks:think 'Hmm' for '2' seconds
Sensing:glide '1' secs to 'Mouse Pointer'
```

Your desktop application should look like **Figure 6**. Now click the green flag at the top of the Stage. You should see the Dog sprite, with a speech balloon that says "You click the mouse and I'll follow." After you click the mouse button, the dog's speech balloon will say "Hmm." Then, two

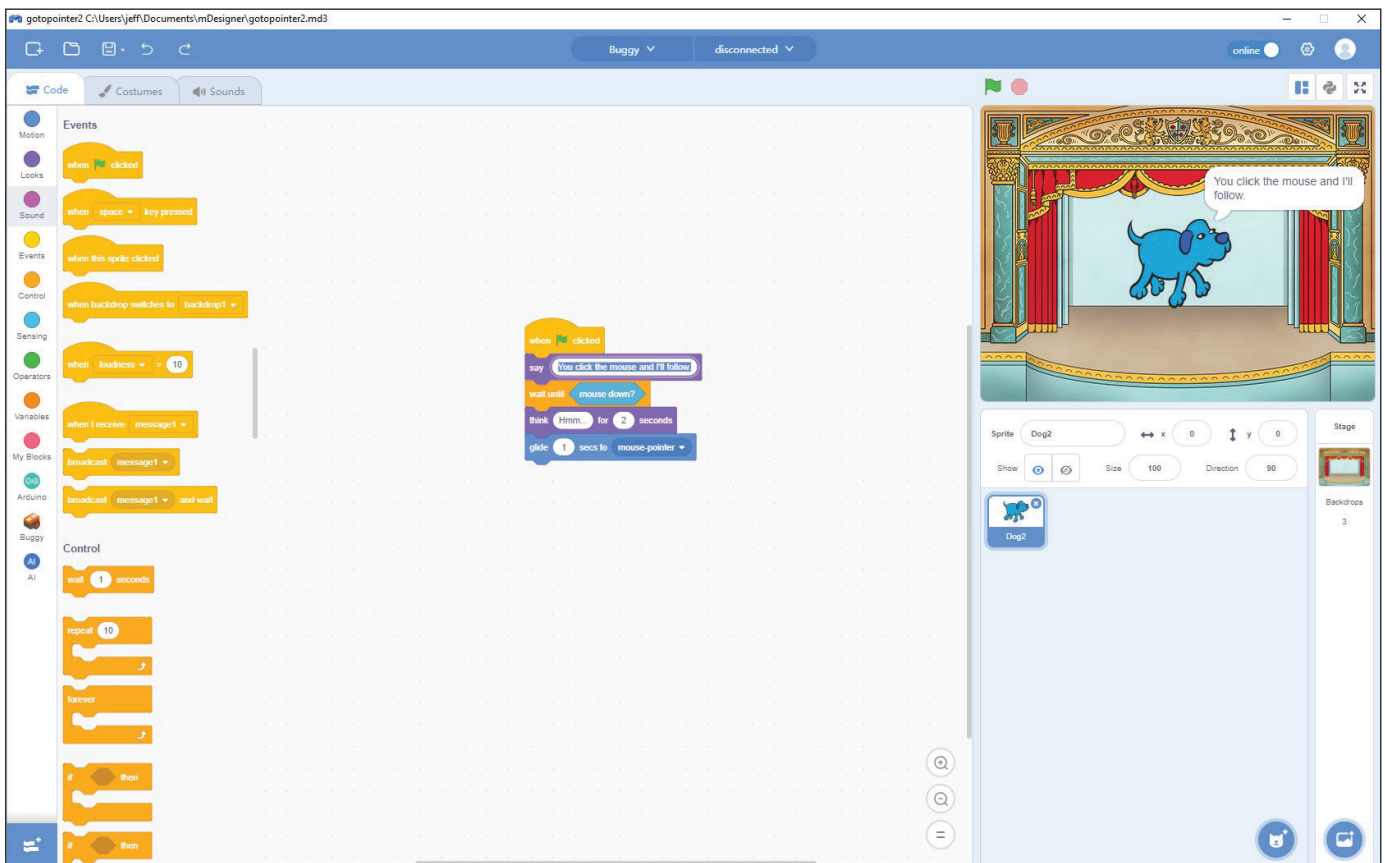


FIGURE 6

Here we've set the stage, so to speak, with our actor—a dog—front and center. Just a few function blocks are required to bring our furry friend to life.

seconds later—after contemplating where the click came from—the sprite will float toward where the mouse pointer was clicked, but will remain inside the Stage area. Add the remaining blocks shown here, and the project is finished.

```
Control:repeat ( )
(Place Operators:pick random '1' to '5' into ( ) )
Sensing:Mouse Down?
(Place previous code into repeat)
Looks:switch backdrop to 'Farm'
Looks:say 'I need a rest!' for '2' seconds
Looks:say 'Zzzzzz' for '10' seconds
(Place each of the following blocks between 'when clicked' and 'say You click the mouse and I'll follow.')
Looks:switch backdrop to 'Theater'
Motion:goto x: '0' y: '0'
Looks:say 'OK, let's play' for '2' seconds
(Place 'Control:forever' around all the code block, with 'when clicked' at the top)
```

Remember the “When Clicked” block must always stay as the topmost block! You should now have what looks like **Figure 7**. Save by clicking “Save Project.” Let your kids play. Fido will tell you when he’s had enough, right? If you haven’t dragged and dropped before, it takes a while to figure out what to grab and where in order to rearrange blocks. There’s a bunch of different codes in this little project. Please feel free to play around. It’s a perfect opportunity to show your kids how easy it is to program. You can build and run at various stages to really get an understanding. Now, on with the show.

BRING ON THE BUGGY

Let’s begin with the difference between “online” and “offline.” When the Buggy is involved, a wired USB connection must always be used. I’m not sure whether the Bluetooth will be incorporated in the future. The online mode will download a small program, which runs on the Buggy and allows your projects to be run/stopped/edited from mDesigner. This is fine if you do not include any movement. Otherwise it’s like having a lively puppy on a leash. The offline mode downloads an executable copy of your project to be run even when unplugged and every time the Buggy is powered up.

If the last project is still loaded into mDesigner, click online to toggle to the offline mode. Two things happen: a notice pops up stating there are blocks that are unsupported in this mode, and on the right side of the screen, the stage and sprite areas are replaced by an area that looks an awful lot like the Arduino IDE! This is about to turn into an educational moment.

Click on “Create New Project” on the toolbar. Add the following blocks to the project:

```
Events:when clicked
Control:repeat '10'
(Place the following inside 'repeat 10')
Buggy:car 'forward' speed '255' duration '1' s
Buggy:buzzer note 'c4' duration '1' s
Buggy:car turn 'left' speed '255' duration '1' s
Buggy:buzzer note 'b4' duration '1' s
```

Note that as you add blocks, the code is generated on the right side of the screen. While mDesigner was always writing code when blocks were added, now you get to see that actual code. This makes it obvious what actual code must be written in an Arduino IDE to perform a described task. Whether or not the user is interested in this presentation, it is an effective learning tool for those interested in tackling the programming hurdle. Save the project and click “flash firmware.” You will need to have the Buggy connected with the included USB cable. If you don’t know which serial COM port to use, run the “Computer Management” application on your PC. On my PC, the port is COM18, as shown in **Figure 8**. When the download is finished, the project will begin running immediately. Note: you may wish to place the Buggy atop something to keep its wheels off the ground. Remove the USB cable and place the Buggy in an open area. It should begin to move and beep, turn and boop 10 times and then stop. The program will run every time you turn on the Buggy with its power switch.

CUSTOMIZING YOUR CODE

You may have noticed while playing with the Buggy and your smartphone that, although you can drive the Buggy around using the joysticks or keep

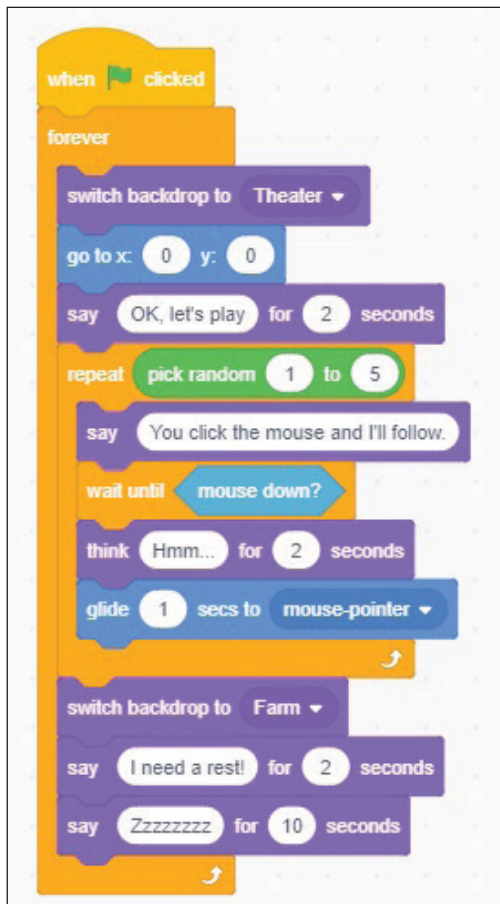
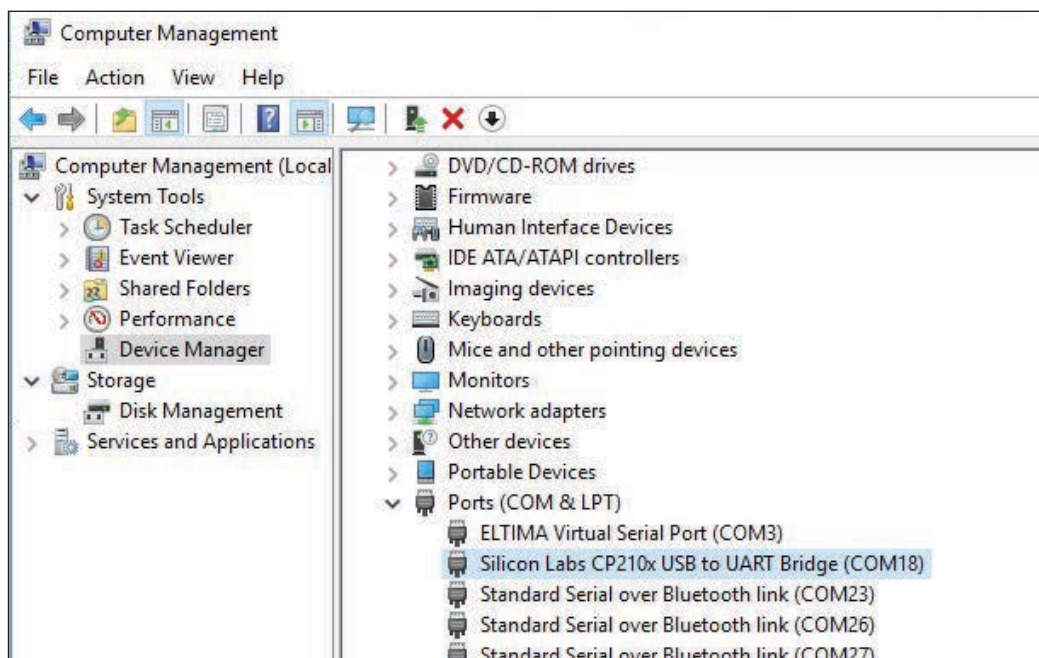


FIGURE 7

Here I’ve expanded on the theme and given the pup a bit of random behavior, along with a change of scenery.

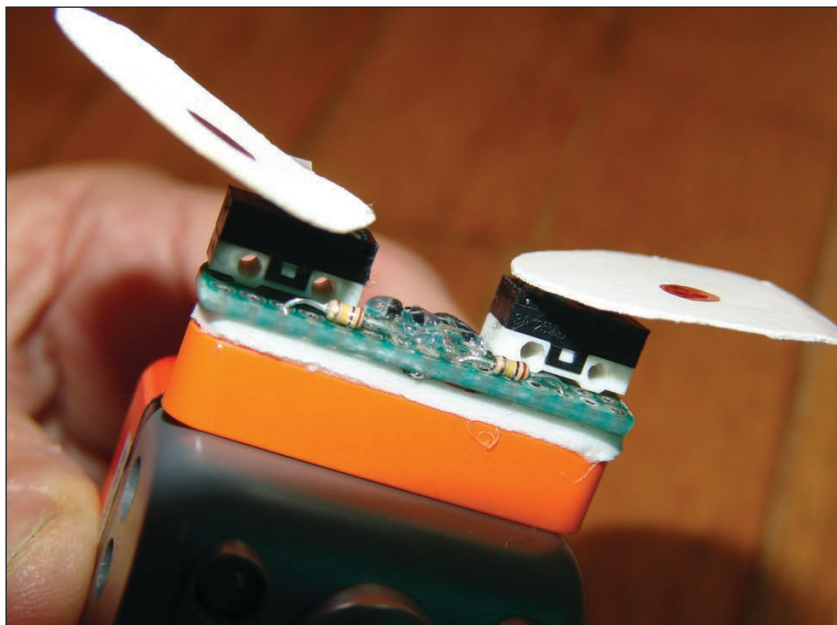
**FIGURE 8**

Under “Windows Administration Tools” you will find “Computer Management”, which shows your active COM ports. If you can’t figure out which one you need, try plugging and unplugging the device, to see what is added or removed from the list.

the Buggy within the playing area when line-following with the color sensor, it has no other input device. I looked at the tiny microswitch offered as a crash sensor and thought “That looks tough to work with. I can do better.” I dug through my “junk” boxes and found some small levered E-Switch microswitches that looked promising. The lever not only reduces the required force to engage the contacts (small, less than 1 oz), but also offers an easy way to affix “crash” extensions.

I grabbed a small piece of protoboard cut to around 0.3" × 1.5"—which is about the size of one of the 1 × 5 orange girders supplied in the Buggy’s extra parts bag. Two of these microswitches and pull-up resistors are mounted on the extreme ends of the PCB. I grounded the N.O. (normally open) switch contacts with the wiper connected to the pull-up and the Buggy input lines of a 4-pin connector. Although I wanted to add the appropriate connector so I could use one of the extra jumpers supplied with the Buggy, I didn’t have one. So, I clipped off one end of a jumper cable and soldered it to the PCB directly. I hope Microduino offers these male connectors for sale in the future for us DIYers.

A piece of double-sided foam tape holds the PCB securely to the girder, which is easily mounted to the front (or rear) of the Buggy. You can see the assembly added to my Buggy in **Figure 9**. Attached to the switch levers are cardboard ovals, which cover the total width of the Buggy and then some for complete protection of most frontal assaults. In robotics, many would say that your sensors should prevent any kind of collision, but even insects use their antennae to make contact with obstacles. Touch is an important

**FIGURE 9**

These tiny lever-action microswitches are just what we need to get this Buggy roaming the range. Arduino blocks let us use external sensors that we connect via the expansion buses.



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.

sensation that we all use every day. The jumper cable is plugged into the Buggy connector marked D4/D5. This means the N.O. switches with pull-ups to V_{CC} look like a logic-level high until an obstacle has closed a switch, which shorts the input to ground (logic low).

FREE TO ROAM

Now we can remove the walls of the Buggy arena and let it roam freely on the range. We've added the hardware to prevent it from getting stuck once its run into a wall, furniture or other obstacle in its landscape. How can we incorporate this sensor into the Buggy's programming? You've probably noticed that the "red" Buggy blocks do not include any mention of any I/O other than motor, color and buzzer. We will need to use the "blue" Arduino blocks, just above the Buggy blocks.

Start by entering the following Buggy block program:

```
Events:when clicked
Control:forever
(Place the following inside 'forever')
Buggy:car 'forward' speed '255' duration '1' s
Buggy:buzzer note 'c4' duration '1' s
```

If you then download the program by clicking "flash firmware," the buggy will begin executing your program. It will drive forward for 1 s before pausing 1 s to play "C4" (that's note C in the fourth musical octave). You can set it free on the floor. Just realize it will eventually crash into something, but attempt to keep going.

A few more lines of code will solve that problem:

```
(Place the following inside 'forever', just under 'buzzer c4 1s')
Control:if '<>'
(Place the following inside '<>')
Arduino:pin '5' is 'low'
(Place the following inside 'if')
Buggy:car 'backward' speed '255' duration '1' s
Buggy:buzzer note 'c5' duration '1' s
Buggy:left 'left' speed '255' duration '1' s
Buggy:buzzer note 'e4' duration '1' s
(Repeat the following, note the minor, but important changes)
Control:if '<>'
(Place the following inside '<>')
Arduino:pin '4' is 'low'
(Place the following inside 'if')
Buggy:car 'backward' speed '255' duration '1' s
Buggy:buzzer note 'c5' duration '1' s
Buggy:turn 'right' speed '255' duration '1' s
Buggy:buzzer note 'g4' duration '1' s
```

Your blocks should now look like **Figure 10**. I saved this project as Buggy Bumpers, and then "flashed the firmware" and set this Buggy free. Note that if the left Buggy switch closes, the Buggy will back up before turning right, away from the obstacle, by executing one "if" statement. A right switch closure should turn the Buggy left (**Figure 11**). If the turns are wrong, that's because your switches are wired or placed the opposite of mine. Then you need to change pins 4 to 5, and pins 5 to 4 in the "if" blocks.

I've included the playing of notes, so you can easily identify which line in the program is executing. You will observe that these "notes" interrupt the movements of the Buggy. To make the movements smoother, eliminate those blocks. Once you do that, you might find that 1 s is too long for some movement—especially the forward movement. Referring back to an earlier project written on the cell phone app, you might say let's shorten them up to some fraction of a second. But fractions of a second are not recognized in mDesigner. Hmm.... A look at the Arduino code (on the left of the screen) written when the blocks are placed will reveal the reason for this.

In the `loop()` code, you'll find the function `BuggyCarTime(1, 255, 1)`—the numbers being values for the variables (direction, speed, duration). We placed the block "car () speed () duration () s" into our project three times. The same function is called each time with different parameters. If you look at this function, which was placed early on in the Arduino code, you find each variable type (`_dir`, `_speed`, and `_time`) has been typecast as `uint8_t`, or an

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

RESOURCES

Microduino | www.microduinoinc.com

E-Switch | www.e-switch.com

8-bit byte (0-255). So, the only legal values that these variables can take on are integers between 0-255. If the provider of the routines had defined `_time` as `float` (floating point), we could have used fractions of a second. You might want to try replacing all the “1”s with 0.1 and see what happens. Then, you can directly alter the Arduino code by replacing the text `uint8_t` with the text `float`, and then recompiling (flashing firmware). Does it act differently? What about the turns? That’s a separate function, but you can replace the `uint8_t` with `float` there also. Don’t forget to recompile.

Note: This is not a permanent change. And because the saved project only saves the blocks, when you reload your project the Arduino code is built again from its libraries. As a result, any changes you made in the code go away. To make this change permanent, you need to do so in the library, where the function is permanently stored.

CONCLUSION


Now you may be able to see that the Itty Bitty Buggy—while an inexpensive and fun “first robot”—is not limiting to your child (or the child in you) in any way. It introduces one to building and operating a number of fun projects that can stimulate one’s imagination and lead to a curious desire to dominate the universe—or at least improve on the program!

A couple things bother me, beyond what I may have already mentioned. Some of the application help is not available in English. However, the Microduino website does have an English Wiki link that is constantly being improved. I’m sure that as the popularity of IBB increases, so will the support.

The sensors available from Microduino were not designed specifically for integration into the Buggy—except for those cookie modules that plug on and have no physical position demands. I think some sensors could be integrated into the mechanical building system, so they could be easily secured to the vehicle/creature being built. Using the tiny connectors may save some real estate, but if they’re going to be made available, a more standard connector would be a plus.

It’s only fair that I praise those areas in which I think Microduino has done things right. The cost of entry is reasonable, and includes many immediately gratifying features. Their redesign of existing hardware has miniaturized what would otherwise be a growing stack of individual products. The inclusion of additional programming environments allows the Buggy’s shelf life to be extended after boredom has set in. The lithium ion battery is extremely important. I’ve seen many abandon any interest in a

device just because low batteries unknowingly introduced erratic operations that didn’t make sense, creating unwarranted frustration.

If you want to provide someone with a fun toy that can encourage learning, I recommend the Itty Bitty Buggy. And don’t forget to add the Buggy Bumper, just to be safe! 

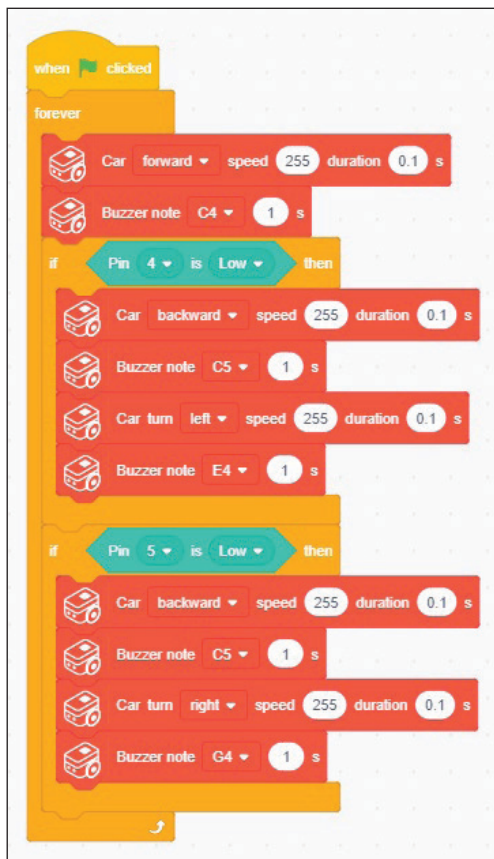


FIGURE 10

With just a few added blocks, our bumper switches can redirect the program flow to back the Buggy away from danger.

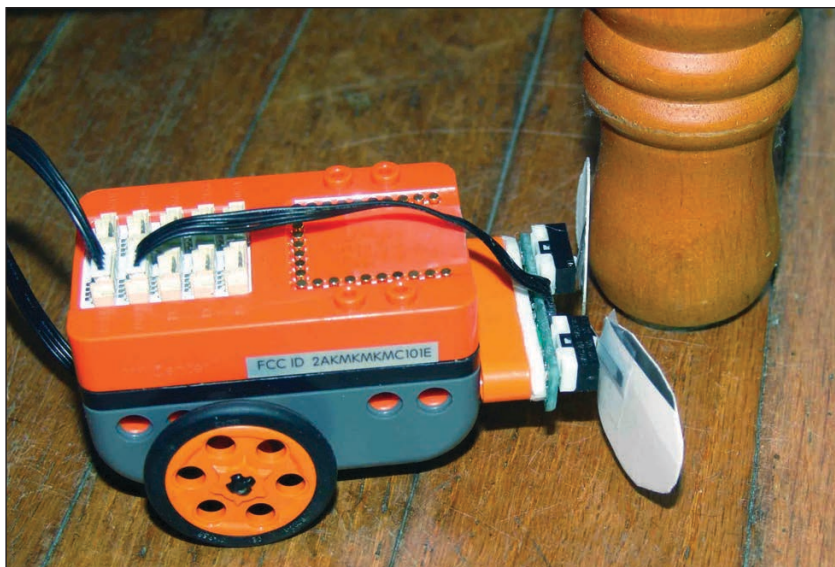


FIGURE 11

My Buggy has run into the leg of a high chair. It’s tough creating avoidance routines that will work for all situations. That’s why more sophisticated robots typically have many levels of avoidance.

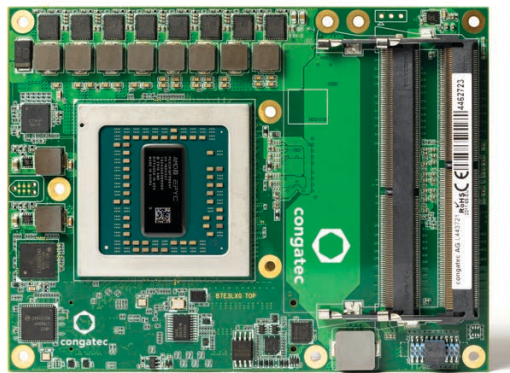
PRODUCT NEWS

COMe Type 7 Card Sports AMD EPYC Embedded 3000 Processor

Congatec has introduced its first Server-on-Module (SoM) with AMD embedded server technology. The new conga-B7E3 Server-on-Module with AMD EPYC Embedded 3000 processor offers up to 52% more instructions per clock compared to legacy architectures, according to the company. Use cases include Industry 4.0, smart robot cells with collaborative robotics, autonomous robotic and logistics vehicles, as well as virtualized on-premise equipment in harsh environments to perform functions such as industrial routing, firewall security and VPN technologies.

The conga-B7E3 COM Express Type 7 modules are equipped with AMD EPYC Embedded 3000 processors with 4, 8, 12, or 16 high-performance cores, support

simultaneous multi-threading (SMT) and up to 96 GB of DDR4 2666 RAM in the COM Express Basic form factor and up to 1 TB in full custom designs. Measuring just 125 x 95 mm, the COM Express Basic Type 7 module supports up to 4x 10 GbE and up to 32 PCIe Gen 3 lanes. For storage the module even integrates an optional 1 TB NVMe SSD and offers 2x SATA Gen 3.0 ports for conventional drives. Further interfaces include 4x USB 3.1 Gen 1, 4x USB 2.0 as well as 2x UART, GPIO, I²C, LPC and SPI. Attractive features also include seamless support of dedicated high-end GPUs and improved floating-point performance, which is essential for emerging AI and HPC applications.



Congatec
www.congatec.com

960 W DIN Rail Supply Boasts 95% Efficiency

TDK has announced the addition of a 960 W rated model to its DRF series of AC-DC DIN rail mount power supplies. The high 95% efficiency produces less internal waste heat enabling electrolytic capacitors to run cooler, providing a calculated life of in excess of eleven years with a 75% load at 230Vac input. The unit can supply a peak load of 1440W (24V 60A) for up to 4 seconds to power capacitive and inductive loads. Applications include industrial process control, factory automation, and test and measurement equipment. The power supply has a 24 V output, adjustable from 24 V to 28 V, using either the front panel mounted trim potentiometer or an external 5 to 6V source. The input range is 180 to 264 VAC, withstanding surges of up to 300 VAC for 5 seconds. The operating ambient temperature is -25°C to +70°C, -40°C cold start, derating linearly above 50°C to 75% load at 70°C.

The DRF960-24-1 is 123.4 mm tall, 139 mm deep and has a narrow 110 mm width saving both space on the rail and in the cabinet. Remote on/off and a 30 V 1 A rated DC OK relay contact are provided as standard. The DRF960 is certified to the safety standards of IEC/UL/CSA/EN 60950-1, UL508 and is

CE marked in accordance to the Low Voltage, EMC and RoHS Directives. The unit is compliant to EN 55032-B (radiated and conducted emissions), EN 61000-3-2 harmonics and IEC 61000-4 immunity standards.

TDK-Lambda | www.tdk-lambda.com



MCU-Based Solution is Qualified with Alexa Voice Service

NXP Semiconductors has unveiled an MCU based voice control solution qualified with Amazon's Alexa Voice Service (AVS). This enables original equipment manufacturers (OEMs) to quickly, easily and inexpensively add voice control to their products, giving their customers access to rich voice experiences with Alexa. Built on an NXP i.MX RT crossover platform, this MCU-based AVS solution enables low latency, far-field, "wake word" detection; embeds all necessary digital signal processing capabilities; runs on Amazon FreeRTOS; and includes an Alexa client application.

This MCU-based AVS solution provides OEMs with a self-contained, turnkey offering that enables them to quickly add Alexa to their products. It includes the MCU, the TFA9894D smart audio amplifier, optional A71CH secure element and comes with fully integrated software. It also features noise suppression, echo cancellation, beam forming and barge-in capabilities that enable use in acoustically difficult environments.

NXP offers at its Mougins, Sophia-Antipolis facilities a product testing service for Alexa Built-in products, available to its customers desiring to test their devices before submitting to Amazon for final evaluation.



NXP Semiconductors | www.nxp.com

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 adcop@circuitcellar.com with your file.

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

LEARN MICRO C ON A PIC[®] CONTROLLER

ONLY \$84.95!

EMBEDDED C PROGRAMMING
Techniques and Applications of C and PIC MCU
Mark Slegemund

The Textbook, Hardware, and Compiler bundle is the perfect combination to achieve a hands-on learning experience for any skill level ranging from beginner to advanced!

Check It Out at www.ccsinfo.com/E3M19

sales@ccsinfo.com (262) 522-6500 X 35

PIC[®] MCU is a registered trademark of Microchip Technology Inc.

ALL ELECTRONICS

Surplus & New Parts & Supplies Since 1967

LEDS · CONNECTORS · RELAYS
SOLENOIDS · FANS · ENCLOSURES
MOTORS · WHEELS · MAGNETS
PC BOARDS · POWER SUPPLIES
SWITCHES · LIGHTS · BATTERIES
and many more items...

We have what you need for your next project.

Discount Prices
Fast Shipping

www.allelectronics.com

Technologic Systems

Single Board Computer

TS-7250-V2
1GHz ARM Computer with Customizable FPGA-Driven PC/104 Connector and Several Interfaces at Industrial Temp

www.embeddedARM.com

Circuit Cellar 2018 Archive

Order yours today

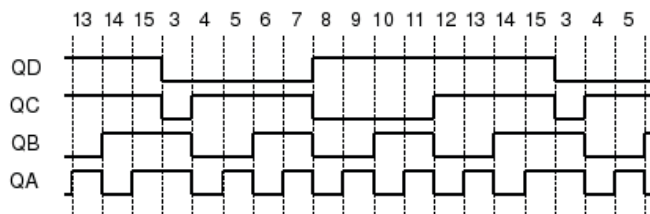
cc-webshop.com

Issues # 330-341 - CD #23
Copyright 2018, XCK Media Corp.

TEST YOUR EQ

Contributed by David Tweed

Problem 1— Back in the days of stand-alone UART chips that required separate baud rate generators, one way to generate the 3.6864 MHz clock for the baud rate generator was to take the 16.000 MHz system clock and feed it to a synchronous 4-bit counter that was configured to divide by



13 by forcing it to load the value 3 when it got to 15, giving the following waveforms:

As you can see, the QB output of the counter produces 3 pulses for every 13 input clocks, and it turns out that this comes very close to the required frequency. What is the exact error, expressed as a percentage?

Problem 2— Obviously, there is some jitter in the timing of the individual pulses produced by this circuit, relative to an evenly-spaced clock at the same frequency. What is the peak-to-peak magnitude of this jitter?

Problem 3— Modern UARTs usually include internal baud rate generators that can divide the input clock by an arbitrary integer N. Given an input clock of 16.000 MHz, and assuming that the output of the baud rate generator needs to be 16x the actual baud rate, what is the highest standard baud rate for which the frequency error is no greater than that generated by the scheme above?

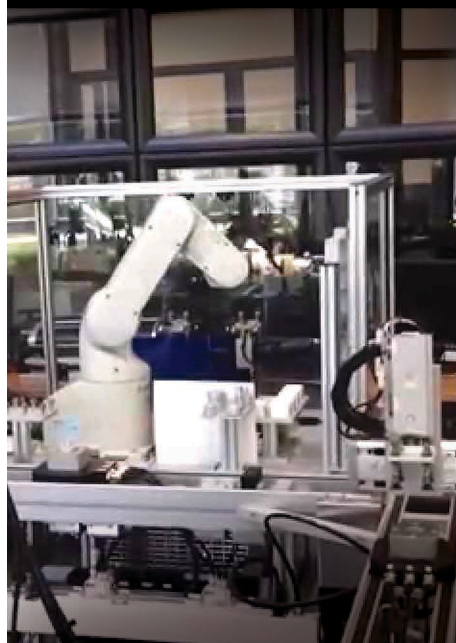
Problem 4— What is the maximum frequency error that a baud rate generator for a UART can produce? Assumptions:

- 8N1 data format
- Error equally distributed between transmitter and receiver
- Generator output is 16x the baud rate

For more information:
circuitcellar.com/category/test-your-eq/

You can take it almost anywhere.

Where will it take you?



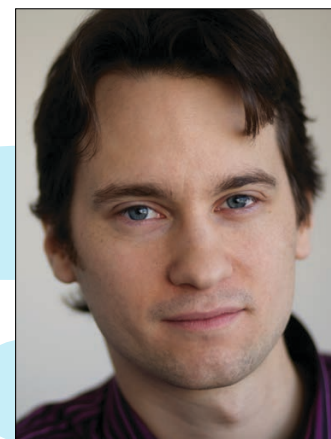
Now you can have the complete Circuit Cellar issue archive and article code stored on a durable and portable USB flash drive.

Includes PDFs of all issues in print through date of purchase.

Visit cc-webshop.com to purchase

The Future of Safe Programming

How Programming Languages Evolve to Reduce Risks



By
Quentin Ochem,
Lead of Technical Account
Management and
Business Development,
AdaCore

The future of microcontroller- and embedded processor-based systems is clear. While there is definitely a large amount of logic that can be directly encoded in the silicon or FPGA, there is also an increasing need for more complex or easier-to-update features to be developed in software on top of it. And these systems may have extremely demanding requirements for safety and/or security. Trends in the automotive domain with assisted or automated driving are a very good example.

The bad news is that programming is much more an empirical process than a deterministic science. Developers can write up to hundreds of lines of code per day—shared with potentially hundreds of other developers—all having slightly different appreciation of what the program should do or how software engineering principles should be applied. Considering that some of these applications will be maintained by generations of developers, it's no surprise to hear that there is roughly one bug per hundred lines of code [1]. Spread over millions of lines running on the simplest device nowadays, this means tens of thousands of lurking bugs, opening doors to hackers and potentially jeopardizing life or property.

Traditional industry response has been a combination of processes and tools. These have been successful, but also come at a cost in terms of verification effort. However, this also comes at a cost in terms of verification effort. As a result, outside of domains such as aerospace and defense, almost no industry has been able to justify the effort.

The tide is turning, though. As software is getting more and more tightly involved in almost every device, so are demands for safety and security. The automotive domain with assisted or autonomous driving is a good example (**Figure 1**). The increased cost may look prohibitive at first, but fortunately, there are other ways to improve the situation: starting by improving the programming language itself.

BEYOND THE INFAMOUS C

A painful number of bug reports are linked to vulnerabilities associated with the C programming language, or C-based relatives such as C++. Many tools exist for no other reason than to try to offset these shortcomings, which can be split into two main categories:

- *Error-prone language definition:* These relate to constructs that may either be ambiguous, or be difficult to interpret. Something like: `int * i; i = i + 1` may mean pointer arithmetic to a careful developer, or be mistaken for an integer increment. While simple issues can be identified through static



FIGURE 1

The automotive domain with assisted or autonomous driving is a good example application where safety and security challenges are becoming more complex and dependent on software.

analysis, their accumulation may eventually render the code very difficult to analyze.

- *Lack of specification capabilities:* While C fully allows a developer to express how a program works, it doesn't provide much capability for expressing what it should do, or under which constraints it should operate. Lack of such specification means many missed opportunities to check software against its intended functionality. External tools exist to work around this issue, but because they're not integrated in the language, they're facing many limitations.

The good news is that there seems to be a renewed effort in the programming language community to provide alternatives to C. For the first issue, an extremely promising example is the Rust language [2] which seems to be getting lot of traction with an imaginative approach to pointer safety. Even within the C community there's a growing understanding that "trusting the developer" shouldn't be a design principle anymore [3].

Initiatives also exist on the second aspect. The "easy" answer is programming by assertion, for example by adding intermediate verification checks in the software that can be enabled or disabled depending on the situation. But there is also progress through the evolution of programming languages. A very good example is the work on the new C++ 202X [4] standard which is looking at extending specification with contracts.

ADA & SPARK: THE NEW WAVE?

There's no doubt that programming languages at large are improving at meeting safety and security concerns. However, a programming language was designed 35 years ago to solve these very issues, and today is still one of the most credible alternatives to C for safety and security purposes: the Ada programming language. Like others, it has come through many revisions, with contract-based programming introduced in its 2012 revision for example. From the start, it has offered a very precise and explicit semantics, and has provided mechanisms for specifying code constraints such as scalar type ranges, array bounds, data mapping on memory, floating point value precision and many others.


Having a precise definition and constraints in the language gave rise to the formally analyzable SPARK subset [5] of Ada. With Ada as with any other languages, processes and tools will be needed. But with a strong foundation, it's possible to go much further. Most C static analysis tools are in the business of identifying potential bugs while being unable to guarantee that all have been caught.

Using the SPARK subset of Ada together with the corresponding tool support can guarantee that all errors of a certain category have been found, with a very low rate of false alarms. Demonstrating a property such as absence of buffer overflow for example becomes a practical matter, opening the door to much more advanced functional analysis, such as proving a program against some of its requirements. This is not just an advantage in theory; an analysis conducted by market research firm VDC in 2018 [6] demonstrated that Ada could lead to cost reduction up to 40% over C in certain industries.

It's therefore no surprise that the Ada and SPARK, which were still a few years ago mostly used within the aerospace domain, are now being adopted by a new wave of users, in industries such as medical devices [7] [8], automotive [9], security [10] and semiconductors [11].

THE DAUNTING LEGACY

There is, however, an important point that all of the above assumes. Adopting Ada or SPARK, going the Rust route or switching to a future version of C or C++ will require you to deal with legacy software, which wasn't written for these standards. While it may be reasonable to take on some minimal rewrite, the cost of rewriting these millions of lines of code will be prohibitive.

As always, there's a reasonable alternative. Most of these languages interface well with legacy C software. With C or C++ this is obvious, but for example, Rust and Ada/SPARK also provide specific support for interfacing with C. So, the idea that the industry is going towards to is: keep the legacy software, rewrite what's highly critical or sensitive, and develop new modules with whichever new language is selected. This will allow the new code to reach proper levels of safety more effectively, giving some more time to find solutions to improving what can't change. 

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

Reference [1] through [11] as marked in the article can be found there.

RESOURCES

AdaCore | www.adacore.com

Quentin Ochem has a software engineering background, specialized in software development for critical applications. He has over 10 years of experience in Ada development. His responsibilities at AdaCore include leading technical account management as well as driving business development, following projects related to avionics, railroad, space and the defense industries.

STRONG FOUNDATION

Whether you are an
EMS, CM or OEM,
let our bare boards be the foundation
you build your reputation upon!

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

**We will make only what is needed,
when it's needed,
and in the amount needed.**

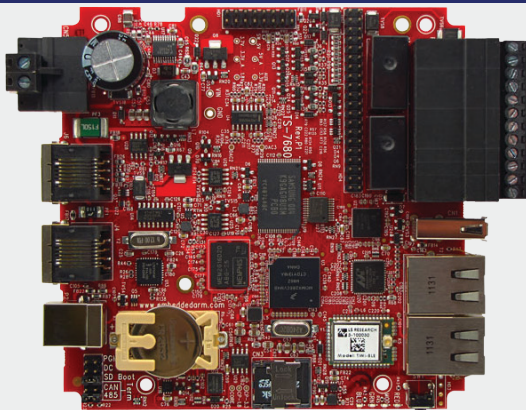
You no longer have to worry about long shelf life
or tie your capital in bare board inventory.

Accutrace[®] inc.

www.PCB4u.com sales@PCB4u.com

SAM & ITAR Registered UL E333047 ISO 9001 - 2008

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 $+85^{\circ}\text{C}$ 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.



Made in USA
with Global Parts

 **Technologic**
Systems