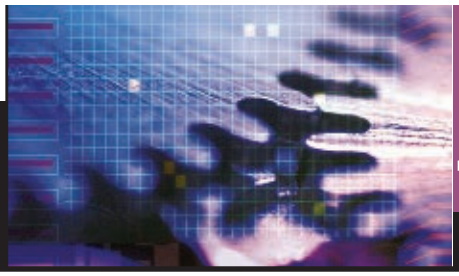


BY FRED EADY

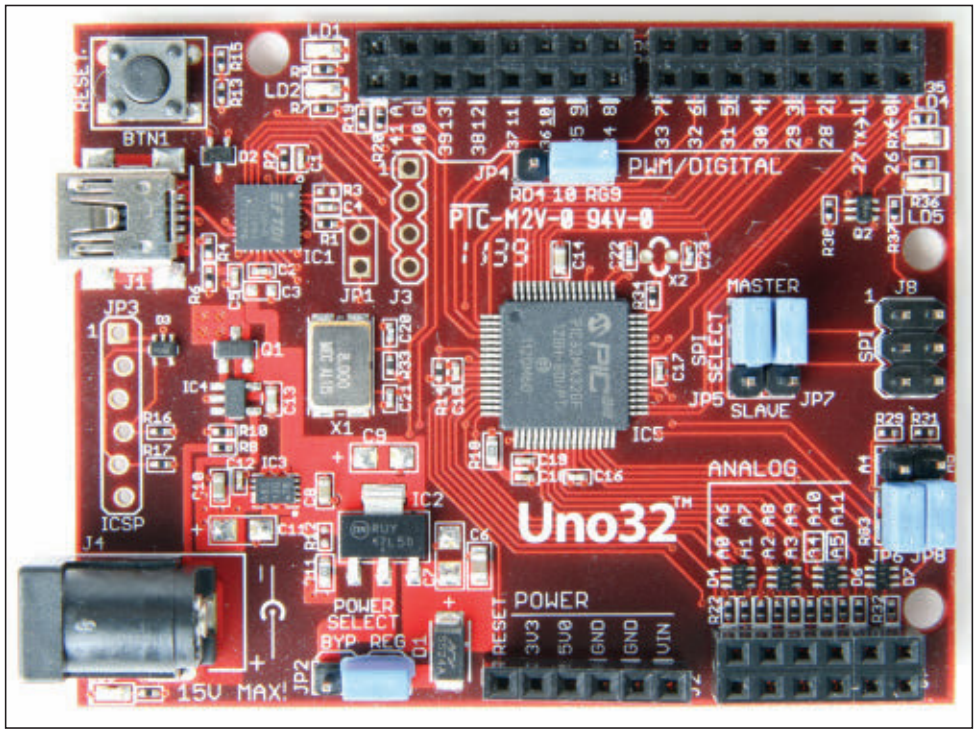
Discuss this article in the *Nuts & Volts* forums at http://forum.nutsvolts.com.
Reprinted by permission of T&L Publications, Inc.

IT'S ALL ABOUT THE Uno32 HARDWARE

It used to be all about the hardware. Remember your first personal computer? Yep, you and I were enthralled and bedazzled by the canned software applications and the command line OS. In reality, our immediate intention was to understand enough about the PC's hardware design to interface with the microprocessor and the I/O subsystem.

I was talking to a friend last night and the discussion fell back into the good old days of the Intel 1702 EPROM and 8085 microprocessor. The 1702 required multiple supply voltages and a unique negative 48 volt programming pulse sequence to store data in its 256 x 8 memory matrix. If your program was big enough, it took a full 30 seconds to program all 256 bytes of the 1702's nonvolatile memory. The 1702 was eventually replaced by the single voltage 1K x 8 2708 and the 2K x 8 2716. I can remember having multiple 2716s that I rotated through the UV eraser with every new spin of code. The EPROMs and companion microprocessor were all we had to work with, and work with them we did. I couldn't afford the expensive EPROM programmers of the time. So, I (and everybody else) designed and built my own. Compilers and printed circuit boards were food for the gods. We used assembler and wire-wrapped breadboards. In the end, it was all about the hardware.

Despite the advent of the instant technology found in our smart phones and electronic notepads, it is still all about the hardware. If it weren't for the 1702, the 8085, and the men and women that plied them, we would still be plunking quarters into public telephones and replacing the needles on our phonographs.

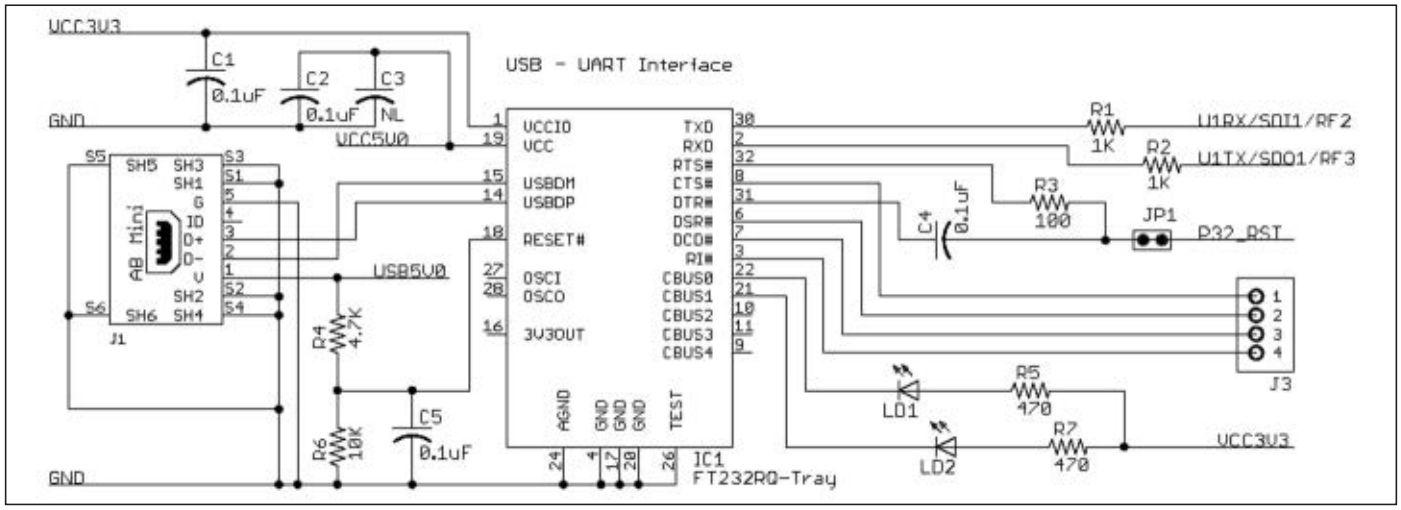


■ PHOTO 1. The Uno32 is a very simple electronic design with very powerful possibilities.

A NEW HARDWARE MODEL

The modern microcontroller has replaced the breadboard mounted rows and columns of combinatorial logic ICs that I used to design around. Gone also are the heavily heatsinked high current power supplies that were required to support the garden of TTL logic ICs.

The discrete hardware components that we can buy from electronic parts distributors are getting smaller and smaller. Some are too small for even the most experienced electronics experimenter to mount



■ SCHEMATIC 1. A MAX232 just wouldn't fit very well in this role. The FT232RQ has it beat in terms of speed and versatility.

by hand. Thus, instead of revolving around a set of passive discrete components, we find ourselves plotting designs that are centered on a cluster of "smart" electronic components.

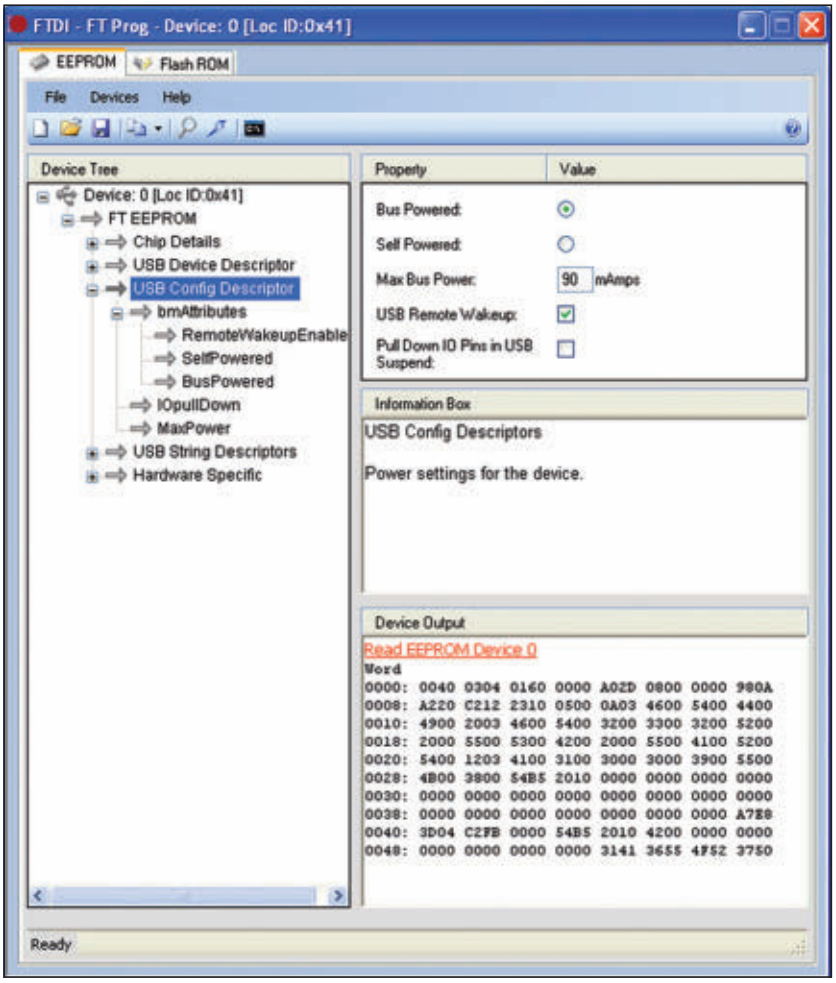
In one-off designs, the passives and logic that made up your electronic cluster would be customized for the application. However, universal electronic cluster designs tend to be cheaper in the long run since the universal cluster can be adapted to multiple missions. More often than not, a universal electronic cluster is built around a microcontroller. The Arduino hardware concept is a perfect example of a universal electronic cluster design.

The Arduino environment provides a way to write applications without the Arduino programmer having to know anything about the hardware. From a pure programmer's point of view, that's a good thing. From a hardware head's point of view, that could be a bad thing. I like to know everything I can about the hardware. Since you're reading Design Cycle, I'm sure you do too.

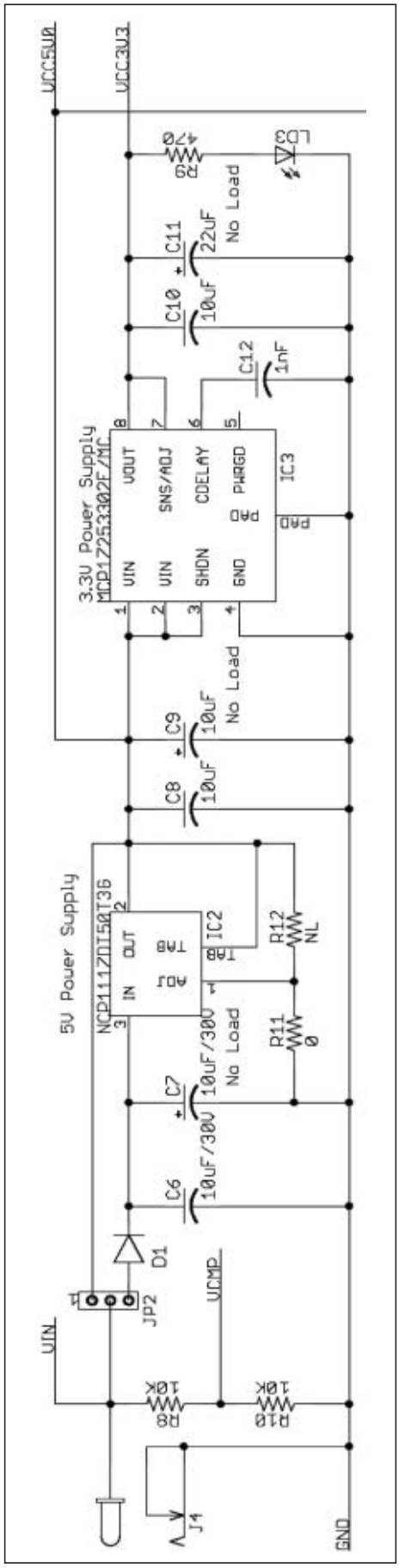
IT'S ALL ABOUT THE HARDWARE

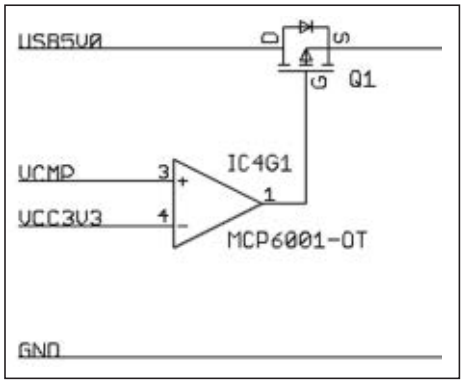
The Digilent Uno32 captured in **Photo 1** qualifies as a universal electronics cluster that can interoperate in an Arduino or native programming environment. I'm sure that there are new Arduino sketches being written for the Uno32 as we speak. While the Arduino programmers are hammering out libraries and applications, we'll turn our attention to the Uno32's 32-bit electronics cluster.

The Uno32's serial port design is graphically depicted in **Schematic 1**. The Uno32 embedded platform uses an FTDI USB device instead of an RS-232 interface IC as its serial port interface hardware. The advantages of implementing a USB interface solution in this situation are



■ SCREENSHOT 1. FTProg is a free utility that can be downloaded from the FTDI site. Here, we see that the Uno32's USB portal is bus powered, asking for a maximum of 90 mA of that power.





■ SCHEMATIC 2. When the MCP6001 op-amp's VCMP input is less than +3.3 volts, MOSFET Q1 allows the Uno32 to be powered by the USB host.

many. For instance, it is not likely that you'll find a nine-pin RS-232 portal on any desktop or laptop computer that is marketed today. However, odds are that you will find at least one USB portal on any PC you can buy new right now. In terms of speed, the FT232RQ easily outruns any MAX232 variant. The FT232RQ is also more versatile than a single RS-232 interface IC.

A single FT232RQ can process transmit and receive data traffic while simultaneously supervising all of the common modem control signals. Transmit/receive visual signaling duty is also handled by the FT232RQ via its programmable five-bit I/O control subsystem. A single FT232RQ literally replaces a herd of RS-232 ICs.

The Uno32's FT232RQ data transfer functions are supported on the PC side by a set of FTDI drivers that are used to construct a Virtual Comm Port, or VCP. On the Uno32 side, the FT232RQ needs no special programming and can be configured using FTDI's FT Prog Utility.

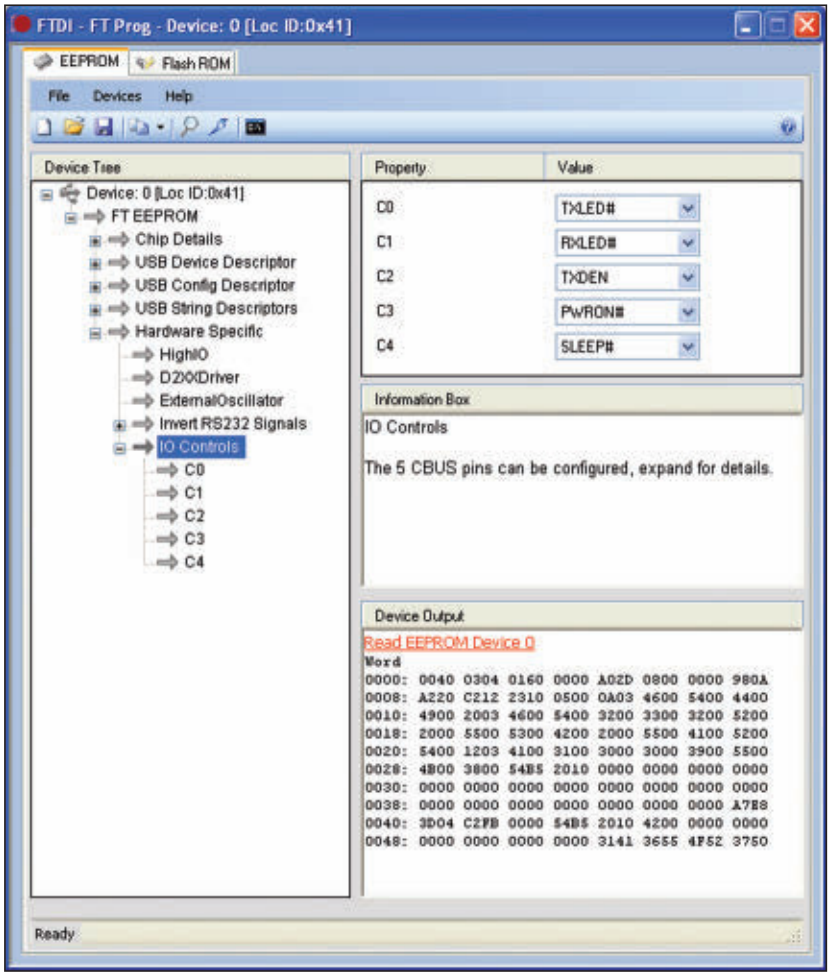
I've connected my laptop to our Uno32's FT232RQ-based USB portal and fired up an instance of the FTDI FT Prog Utility. **Screenshot 1** contains the USB Config Descriptor information. The **Screenshot 1** FT232RQ scoop tells us that the Uno32's USB portal is bus powered and will require a maximum of 90 mA from the USB host. The fact that the Uno32's FT232RQ is bus powered is not obvious in the realm of **Schematic 1**. So, let's follow the FT232RQ's power train beginning with **Schematic 2**.

The MCP6001 operational amplifier you see in **Schematic 2** is used as a comparator. When the MCP6001's VCMP input is presented with less than +3.3 volts, MOSFET Q1 is biased ON and allows the USB host to power the Uno32 via the USB host portal. MOSFET Q1's source pin feeds the VCC5V0 power signal you see in **Schematic 3**. While your eyes are on **Schematic 3**, note that the Uno32 can also be powered by an external power source other than a host USB portal. The VCMP power signal is derived from the external power source that does not originate from the USB host. Thus, the use of a USB host power source and an external non-USB power source is mutually exclusive. The host USB power source will always be overridden by an external power source greater than 7.0 VDC.

Now that we know how the Uno32's FT232RQ is powered, we can look back at **Schematic 1** and see that the FT232RQ's I/O subsystem is set up for 3.3 volt logic levels via the voltage applied to its VCCIO pin. Also, note that the FT232RQ's active-low RESET pin is controlled by the host USB portal. The voltage divider made up of resistors R4 and R6 provide a pullup voltage source for the internal 1.5KΩ pullup resistor that is electrically attached to the FT232RQ's USBDP pin. When power is applied to the internal pullup 1.5KΩ resistor, the USB host is informed by the presence of the pullup voltage that the Uno32's FT232RQ is a full speed USB device.

Moving to the modem control / I/O side of the FT232RQ, we find transmit and receive data signals, a modem control instantiated CPU reset signal, and visual indicators in

■ SCHEMATIC 3. No rocket science here. If you've ever worked with a linear voltage regulator, everything should be obvious to even the most casual observer.



■ SCREENSHOT 2. We have no direct control of LEDs LD1 and LD2 unless we reconfigure the FT232RQ's I/O controls.

the guise of a couple of LEDs. The FT232RQ's TXD and RXD pins are operating at 3.3 volt logic levels and are electrically attached to the Uno32's PIC32MX320F128H UART1 interface.

Driving the FT232RQ's DTR signal logically low presents a reset condition to the PIC32MX320F128H. As you've probably already ascertained, the modem control signals (DTR, RTS, etc.) are under the control of the PC program that's in charge of the data flow on the USB portal.

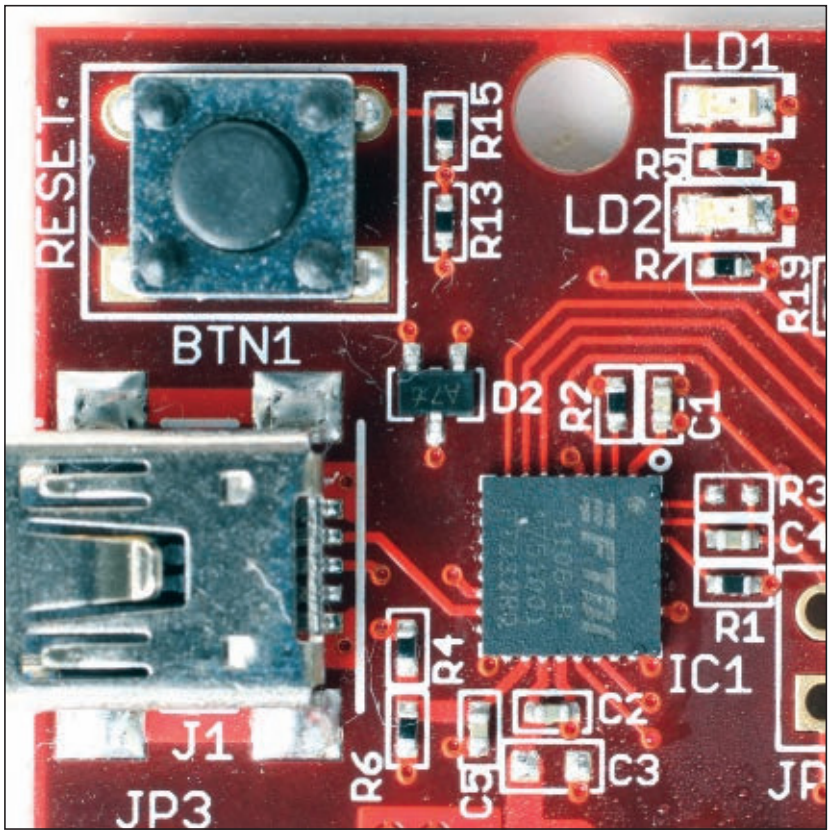
The operation of the LED indicators, LD1 and LD2, is preconfigured using the FTDI FT_Prog Utility. **Screenshot 2** is representative of how the Uno32's FT232RQ I/O controls are set up. LD1 flashes on transmit events while LD2 blinks when data is received.

Photo 2 is a spy satellite view of the Uno32's FT232RQ resident area on the Uno32's PCB. The entire FT232RQ design fits within the space of a quarter. The FT232RQ USB-to-TTL services could also be performed by the PIC32MX320F128H's internal USB engine. The FT232RQ is a better choice here as it conserves PIC32MX320F128H I/O pins by providing the CPU reset and visual indicator control via its software-controlled modem control signals and five-bit I/O control subsystem. Plus, we didn't have to write a byte of code to partake of the FT232RQ's control and signaling services.

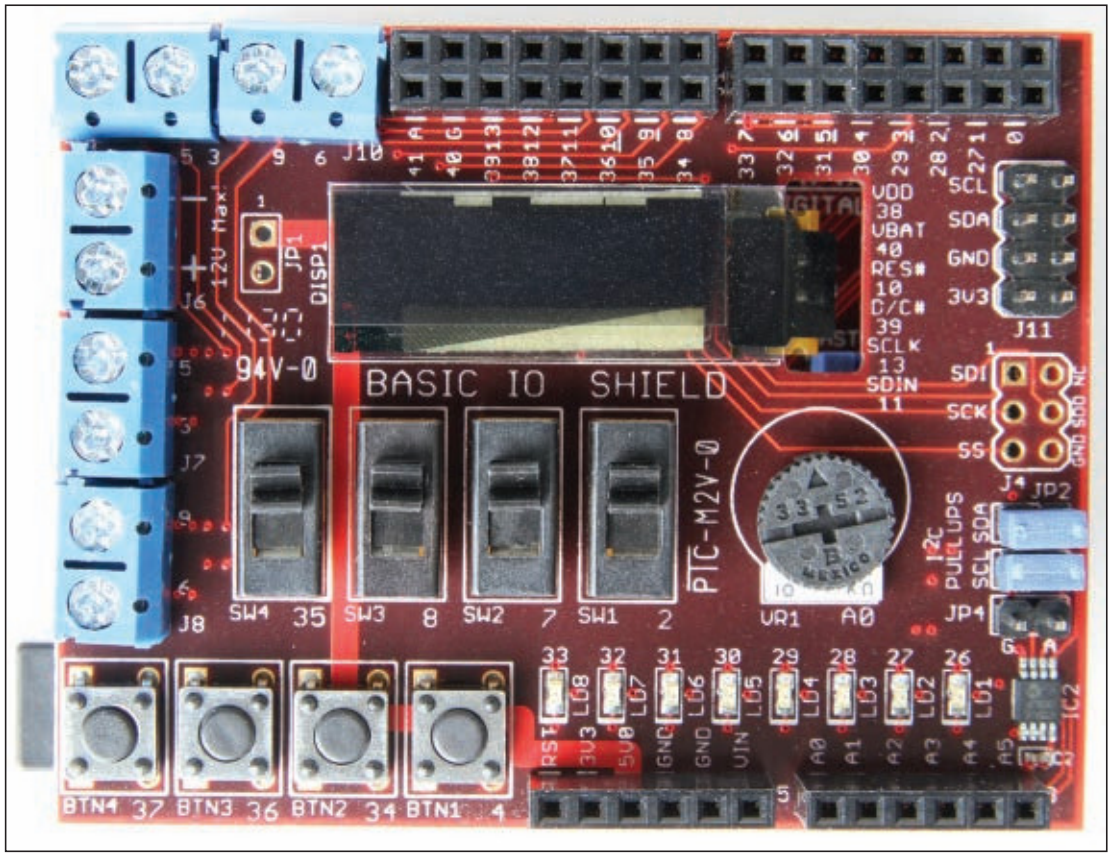
Take another close look at **Photo 1**. The only other programmable electronic component in the universal electronic cluster called Uno32 is the PIC32MX320F128H. All of the Uno32's base peripherals are stationed in the PIC32MX320F128H silicon. I won't quote the PIC32MX320F128H datasheet here since you can download your own copy. All of that peripheral potential energy cooped up in the PIC32MX320F128H silicon is useless if we can't access it. It's all about the hardware. So, let's provide the PIC32MX320F128H's internal peripherals with a playmate.

HUMAN INTERFACE HARDWARE

The Uno32 can be enhanced with yet another electronic cluster in the form of a Digilent Basic I/O Shield. In the Arduino sense,



■ PHOTO 2. Small size and minimal board space requirements are two advantages to the FTDI USB serial interface solution.



■ PHOTO 3. The combination of the Basic I/O Shield and Uno32 is simple enough for the beginner to understand and complex enough to aid in professional embedded development.

the Basic I/O Shield is a universal electronic cluster as it is designed to be interoperable with the stock Arduino hardware. The Basic I/O Shield is made up of familiar electromechanical and electronic components. You can identify most of the Shield's features just by observing them in **Photo 3**.

The Basic I/O Shield is intended for use by both neophyte and professional. Resident Shield hardware includes pushbuttons, slide switches, a byte-wide group of LEDs, and a potentiometer. The potentiometer is designed in as an analog input device and is electrically connected to one of the PIC32MX320F128H's analog inputs.

The Shield also offers advanced I/O devices. You'll find a 24LC256 EEPROM and a TCN75 digital temperature sensor on its I²C bus. The Shield's SPI portal directly supports a 128 x 32 pixel OLED display. For those of you that wish to control high current loads such as motors or solenoids, the I/O Shield is equipped with a quad of N-channel MOSFETs in an open drain configuration.

The idea behind the Basic I/O Shield is to allow the application designer to use standard Arduino calls to manipulate the Shield's I/O devices. To that end, you'll find documentation detailing its pinout in Arduinoese. If you prefer not to go with Arduino as a development tool, you can use the Shield schematic to map the I/O devices to the PIC32MX320F128H. The folks at Digilent do a very good job at documentation, and you can get the information and specifications you need for the price of a free download.

TACOS

There are beef tacos, pork tacos, and fish tacos just to name a few. No matter what's inside the taco shell, it's still a taco. High level languages are like tacos. For instance, the C programming language basically uses the same statements and rules to create assembly code for an 8085 microprocessor as it does for a PIC. In the case of the Uno32, Arduino takes the taco concept one shell further. An Uno32 Arduino language statement begins as a PIC C function. The PIC C function is then wrapped up as a C++ object that eventually becomes an Arduino statement. So, one could say that even with high level languages it's still all about the hardware. To emphasize the point, let's see what's behind the Uno32 Arduino statement that initializes the OLED display that is integral to the Basic I/O Shield.

Here's the Arduino way to wake up the Shield's OLED display:

```
void setup()
{
  IOShieldOled.begin();
}
```

Right away, we know that an object called *IOShieldOled* has been instantiated. We also know that the *begin()* function is a member of the object *IOShieldOled*. The class declaration from which the object *IOShieldOled* is spawned looks like this:

```
class IOShieldOledClass
{
  private:

  public:
    /* Class Constants*/
       static const int colMax = 128;
//number of columns in the display
       static const int rowMax = 32;
//number of rows in the display
       static const int pageMax = 4;
```

```
        //number of display pages
        static const int modeSet = 0;
        //set pixel drawing mode
        static const int modeOr  = 1;
        //or pixels drawing mode
        static const int modeAnd = 2;
        //and pixels drawing mode
        static const int modeXor = 3;
        //xor pixels drawing mode

    IOShieldOledClass();

        /* Basic device control functions.*/
        void begin(void);
        void end(void);
        void displayOn(void);
        void displayOff(void);
        void clear(void);
        void clearBuffer(void);
        void updateDisplay(void);

        /* Character output functions.*/
        void  setCursor(int xch, int ych);
        void  getCursor(int *pxcy, int *pych);
        int   defineUserChar(char ch, BYTE
        *pbDef);
        void  setCharUpdate(int f);
        int   getCharUpdate(void);
        void  putChar(char ch);
        void  putString(char *sz);

        /* Graphic output functions.       */
        void  setDrawColor(BYTE clr);
        void  setDrawMode(int mod);
        int   getDrawMode();
        BYTE* getStdPattern(int ipat);
        void  setFillPattern(BYTE *pbPat);

        void moveTo(int xco, int yco);
        void getPos(int *pxco, int *pyco);
        void drawPixel(void);
        BYTE getPixel(void);
        void drawLine
        (int xco, int yco);
        void drawRect
        (int xco, int yco);
        void drawFillRect
        (int xco, int yco);
        void getBmp
        (int dxco, int dyco,
        BYTE *pbBmp);
        void putBmp
        (int dxcp, int dyco,
        BYTE *pbBmp);
        void drawChar(char ch);
        void drawString(char *sz);
};
```

Everything — I mean everything — that has to do with controlling the Shield's OLED display is defined within the *IOShieldOledClass* class declaration. Note that our *begin()* control function is contained within the *IOShieldOledClass* class declaration's braces. The class declaration can be found in the *IOShieldOled.h* file.

An instance of the class *IOShieldOledClass* called *IOShieldOled* is created in the *IOShieldOled.cpp* C++ source file. We also place C++ code to flesh out the function members of the class *IOShieldOledClass* in the *IOShieldOled.cpp* C++ source file. All of the functions are addressed. However, we're only interested in the *begin()* control function:

```
void IOShieldOledClass::begin(void)
{
        OledInit();
}
```

As it turns out, the *OledInit()* function is part of the Oled driver code which is found in the file *Oleddriver.c*:

```
void OledInit()
        {

        /* Init the PIC32 peripherals used to
        ** talk to the display. */
        OledHostInit();

        /* Init the memory variables used to
        ** control access to the display. */
        OledDvrInit();

        /* Init the OLED display hardware. */
        OledDevInit();

        /* Clear the display. */
        OledClear();

}
```

Since it's all about the hardware, let's look at the meat of the *OledHostInit()* function:

```
void OledHostInit()
        {
        #if defined (_BOARD_UNO_)
                /* Initialize SPI port 2.*/
                SPI2CON = 0;
                SPI2BRG = 4;
                //8Mhz, with 80Mhz PB clock
                SPI2STATbits.SPIROV = 0;
                SPI2CONbits.CKP = 1;
                SPI2CONbits.MSTEN = 1;
                SPI2CONbits.ON = 1;
        #elif defined (_BOARD_MEGA_)
                /* Initialize pins for bit bang
                **SPI. The Arduino Mega boards,
                ** and therefore the Max32 don't
                **have the SPI port on the same
                ** connector pins as the Uno. The
                **Basic I/O Shield doesn't even
                ** connecto to the pins where the
                **SPI port is located. So, for
                ** the Max32 board we need to do
                **bit-banged SPI.
                */
                PORTSetBits(prtSck, bitSck);
                PORTSetBits(prtMosi, bitMosi);
                PORTSetPinsDigitalOut(prtSck,
                bitSck);
                PORTSetPinsDigitalOut(prtMosi,
                bitMosi);
        #else
                #error "No Supported Board
                Defined"
        #endif
        PORTSetBits(prtDataCmd, bitDataCmd);
        PORTSetBits(prtVddCtrl, bitVddCtrl);
        PORTSetBits(prtVbatCtrl, bitVbatCtrl);

        PORTSetPinsDigitalOut(prtDataCmd,
        bitDataCmd);  //Data/Command# select
        PORTSetPinsDigitalOut(prtVddCtrl,
        bitVddCtrl);  //VDD power control (1=off)
        PORTSetPinsDigitalOut(prtVbatCtrl,
        bitVbatCtrl); //VBAT power control
        (1=off)
```

```
        /* Make the RG9 pin be an output. On the
        ** Basic I/O Shield, this pin is tied to
        ** reset.*/
        PORTSetBits(prtReset, bitReset);
        PORTSetPinsDigitalOut(prtReset, bitReset);

}
```

We're finally down to the C code that speaks directly to the PIC32MX320F128H. Every function within the *OledInit()* function will execute initializing the PIC32MX320F128H, the Oled hardware, and the Oled driver. All of the definitions for the arguments of the functions contained within the functions found in *OledInit()* are contained within the *OledDriver.h* file. The *PORTSet* functions are PIC32MX peripheral library calls that are part of the PIC32MX Peripheral Library, which is part of the Microchip.PIC32MX C compiler.

IT'S STILL ALL ABOUT THE HARDWARE

Now you know what happens behind that *IOShieldOled.begin()* Arduino statement. I guess you could classify the Arduino statement we examined with one of those tacos with two shells separated by cheese. The C and C++ code we discussed ends up being compiled to form an Arduino OLED library. Naturally, the *IOShieldOled.xxx* calls are used in your Arduino sketches.

If we equate eating with coding, Arduinoese is easy to swallow. The C behind the Arduinoese is a bit chewy, but easily consumed. If you're interested in moving from being an Arduino user to an Arduino developer, the Uno32 can help you get there. The Uno32 comes with a factory installed bootloader that is designed to be used with Digilent's MPIDE Arduino-compatible development tool. For those of you that aren't grazing on the Arduino farm, the Uno32 is PICkit3-ready too.

You can use assembler, C, C++, and Arduinoese to prod the Uno32's PIC32MX320F128H into programmable action. The bottom line is that the Uno32's hardware is a common denominator no matter what programming language you choose to use. The Uno32 puts 32-bit Arduino computing into your Design Cycle. **NV**

SOURCES

Digilent
Uno32
Basic I/O Shield
MPIDE
www.digilentinc.com