

■ BY FRED EADY

www.nutsvolts.com/index.php?/magazine/article/august2011_DesignCycle

# GIVE YOUR BITS SOME AIR

**If you think that you need ZigBee or 802.15.4 to move small chunks of data with a low power data radio, you are absolutely correct. If you think that you don't need ZigBee or 802.15.4 to move small chunks of data with a low power radio, you are absolutely correct.**

**ZigBee and 802.15.4 are wireless data communication standards that require packet-descriptive information to be sent along with the packet's payload data. If you only need to send a couple of bytes in a peer-to-peer or multicast environment, you really don't need (or want) the network overhead that comes with an official ZigBee stack or 802.15.4 network.**

When NASA put men on the moon, there was no personal computer for the masses, no USB, no Ethernet (as we know it), no Internet (as we know it), no 802.15.4, and no ZigBee. In addition to these shortcomings, the RS-232 standard was in its infancy. Yet, NASA still managed to get one of the world's largest rockets, three men, a command module, a service module, and a lunar module to make the 500,000 mile round trip. For those of you born in the 1970s and beyond, the lunar module and the Saturn V booster only made half of the trip or less. Apollo 13 was the exception as the lunar module was retained and used on the return leg to get the astronauts (along with the crippled service module) safely back to Earth. Lacking today's technology, it's a certainty that NASA used sophisticated one-off computing devices and radio equipment to fly manned and unmanned missions in the early days of the space program. In the spirit of 1969 technology, I'm going to show you how to wirelessly transfer data between multiple nodes without resorting to ZigBee, 802.15.4, specialized computers, or one-off radios.

In this installment of Design Cycle, we're going to use a modular approach to construct some embedded data radio hardware. Once the hardware comes online, we'll put on our software hats and spin some code. When we finish the coding, we'll dump the bits into the hardware, put on our pointy hat that is decorated with stars and moons, and observe data magically move from one radio platform to the other.


■ PHOTO 1. The ultra-compact size and low power consumption of the AIR module make it an ideal radio platform for low power wireless embedded projects.

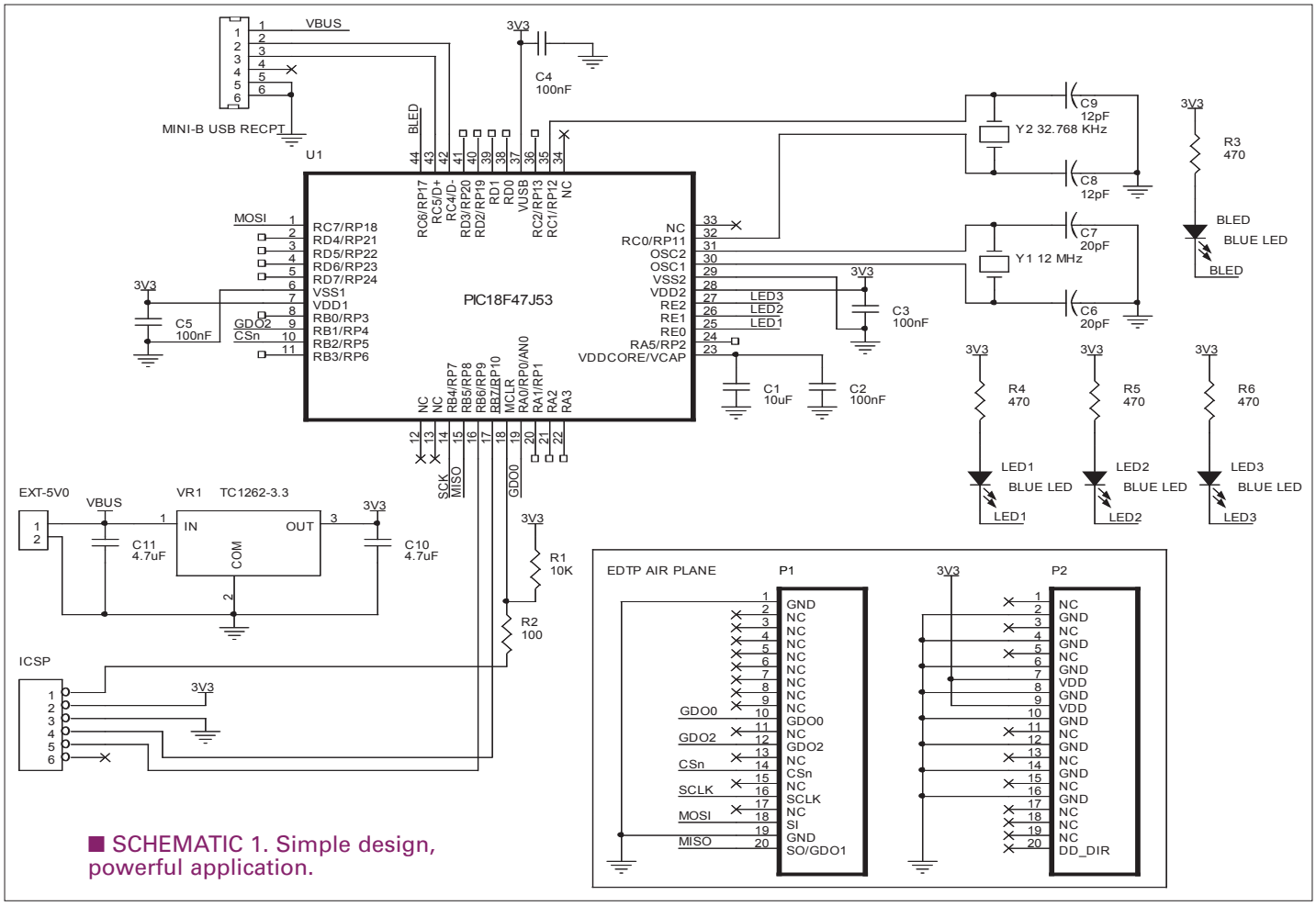
## GETTING ON THE AIR

In this case, AIR is short for Anaren Integrated Radio. The 2.4 GHz AIR modules measure in at 9 x 12 x 2.5 mm. The sub-postal stamp-sized AIR module you see in **Photo 1** houses an integrated crystal, a voltage regulator, and all of the associated RF circuitry necessary to support its CC2500 transceiver core. In idle mode, the AIR module draws a paltry 1.5 mA. When sleeping, the 2.4 GHz module current requirement drops to a nearly nonexistent 400 nA. Anywhere from 13.3 mA to 19.6 mA current draw is typical in receive mode, and at maximum transmit power only 21.5 mA is consumed. This level of power consumption allows the AIR module to fit nicely into very low power telemetry and embedded control applications.

The 2.4 GHz A2500R24A AIR module pictured in **Photo 1** is equipped with an integral antenna. If your module is to be imprisoned in a metal enclosure, you'll need the A2500R24C variant which is fitted with a compact U.FL antenna connector. This month, we will work exclusively with the A2500R24A module. So, from now on the A2500R24A will be referred to simply as the AIR module. The AIR module is documented extensively on the Anaren website. So, there's no need to rehash in this text the information you can easily access online. The main goal this month is to design, assemble, and code a microcontroller-based support system for the AIR module.

## AIR SUPPORT

Just because NASA sent men to the moon sans USB doesn't mean that we can't employ the services of USB in our AIR design process. In the firmware debug phase of the design process, we'll use USB as a power source and as a stand-in for RS-232. That implies that our microcontroller of choice must be USB capable. The next design point our microcontroller must meet concerns the compiler we will use to forge the AIR module firmware driver. We will need to select a compiler that supports USB. Access



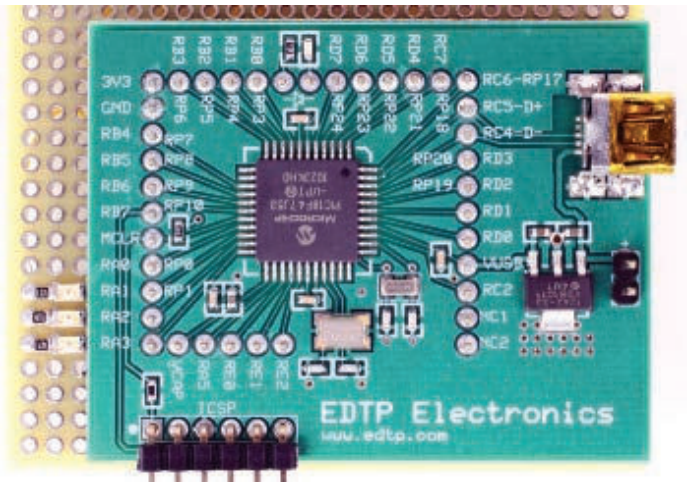

■ SCHEMATIC 1. Simple design, powerful application.

to the AIR module's internal registers is enabled via a four-wire SPI portal. The SPI protocol can be emulated with user-written bit bang routines. So, our compiler need doesn't built-in SPI functionality, but it would be nice if it did. We don't have to look far for a suitable microcontroller. The PIC18F47J53 natively supports USB and offers an on-chip hardware-based SPI engine. We'll be storing tables in Flash and data in buffers carved from SRAM. The PIC18F47J53 has ample memory resources that we can call on. On the power plane, the PIC18F47J53 is a perfect fit for the AIR module as both it and the PIC18F47J53 operate on a 3.3 volt power rail. The equality at the power plane level eliminates the need for logic level shifting of the PIC18F47J53's SPI portal and I/O pins. Thus, we can directly connect the I/O subsystems of the PIC18F47J53 and AIR module. I would like to use the CCS C compiler as it runs under the influence of MPLAB and allows the use of Microchip's PICkit3 as a debugger and programmer. As it turns out, the CCS C compiler also fully supports the PIC18F47J53's USB and SPI engines.

The PIC18F47J53 design is chronicled in **Schematic 1** and realized in **Photo 2**. A 32.768 kHz crystal is included in the PIC18F47J53 design to enable the PIC's internal RTCC (Real Time Clock Calendar). The option to employ the PIC18F47J53's analog-to-digital converter (ADC) is available by way of the free PORTA I/O pins. I've included some optional debug/status LEDs on the PORTE pins. If you need the PORTE pins as additional analog-to-digital inputs, you can move LED1-LED3 to other available PIC18F47J53 I/O pins or eliminate them all together. The PIC's USB portal doubles as a power source and a USB CDC (Communications Device Class) device. A companion CCS USB driver on the PC side sets up a virtual COM port that allows the PIC18F47J53 to communicate with a terminal emulator using its embedded USB portal.

## THE AIR PLANE

Now that we have the microcontroller host portion of the project under control, we can begin work on the AIR frame. The AIR module grinning at you in **Photo 1** is an SMT device that is more suited to projects that have been tested and finalized. The Anaren engineers that wear those pointy hats brewed up yet another AIR module variant. The A2500R24A-EM1 was originally conceived to allow AIR modules to ride on Texas Instruments SmartRF evaluation boards. We're going to hijack the A2500R24A-EM1 tied up in **Photo 3** and put it on another plane. With a little help from our friends at SAMTEC and ExpressPCB, I whipped up the AIR PLANE which is basking in the light of **Photo 4**. The AIR PLANE is a hardware conversion tool that pulls the SAMTEC-based 40-pin A2500R24A-EM1 interface into eight pins that are placed on convenient 0.1 inch centers. Contained within the AIR PLANE's eight-pin interface are the four-wire SPI portal, the AIR module GDO0 and GDO2 I/O pins, and power and ground points. The AIR PLANE's converted interface is everything we need to fully access the AIR module's configuration and data registers.




■ PHOTO 2. The PIC18F47J53 hardware shown here is designed to drive an AIR module in stand-alone mode using battery power, or to control an AIR module under the influence of a PC's USB portal.

As you can see in **Photo 5**, all of the modular hardware components are mounted in the 0.1 inch pitch fiberglass grid of an EDTP plated-through perf board. The short ends of standard 0.1 inch pitch male headers are soldered on the component side of the PIC18F47J53 module. The extended portions of the male headers are long enough to pass through the perf board and act as wire wrap posts. I chose to use a female header to socket the AIR PLANE and its A2500R24A-EM1 evaluation board cargo.

## GETTING AIRBORNE

I applied power to the collaboration of modules assembled in **Photo 5** and did not release any magic smoke. So, we're ready to put down a firmware foundation that will allow us to take our AIR module and supporting equipment down the runway, and ultimately go AIRborne. To those RF types that wear the pointy witch hats, the AIR module is a collection of well-placed coils and capacitors that transfers data by disrupting small portions of the Earth's magnetic field. To a hardware type, the AIR module is a tiny building block that sits on a particular layout of printed circuit board (PCB) pads. To a programmer, the AIR module is a logical collection of registers and FIFO (First In First Out) buffers. All of the RF plumbing has been done for us by the folks at Anaren. We took care of the AIR hardware build ourselves with the fabrication, assembly, and integration of the PIC18F47J53 and AIR PLANE modules. There's no one else here to fly this thing but us. So, let's start flipping software switches and see if we can't get this baby on the AIR.

## A SOFTWAIR RADIO

My very first serious radio was a Knight Kit Ocean Hopper shortwave radio (**http://nostalgickitscentral.com/allied/products/knight_radio.html**). I recall the various coils that could be plugged in for listening in at different frequencies. The in-band tuning was done mechanically via a dial that was attached to a large variable capacitor. The only software involved in tuning the Ocean Hopper were my — at the time — itty bitty little fingers.

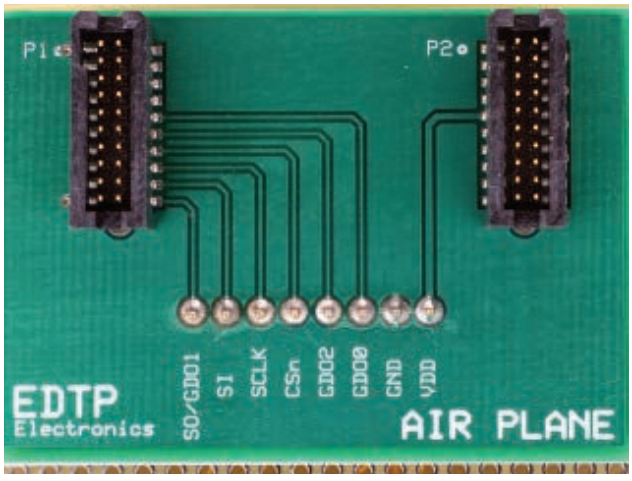
A couple of HC49-packaged microcontroller crystals would drown the tiny AIR module. Obviously, as far as the AIR module is concerned, there's not enough real estate available for any mechanical RF controls. Thus, the AIR module is a fly-by-wire device. Registers and the values contained within them replace the variable capacitors and frequency selection coils. In the case of the AIR module, all of its RF and data handling parameters are controlled by the contents of the 47 registers enumerated in **Listing 1**.

After staring a hole into the front panel of my Ocean Hopper, I longed for some visual feedback on the signals I was receiving in my headphones. Back in the day, the more sophisticated shortwave receivers came equipped with signal strength meters. Although a mechanical meter could be electrically adapted to the AIR module, an external mechanical or electronic metering device would be overkill as the AIR module has a built-in set of digital meters in the guise of status registers. The set of digital status meters are contained within the register set you see in **Listing 2**.

**Listing 2** also exposes the AIR module's PATABLE and FIFO registers. The PATABLE consists of eight bytes and is instrumental in dialing in the AIR module's transmit output power. Our PATABLE setting looks like this:

```
//*********
//* AIR
//*PATABLE SET
//*FOR 0dBm
//*********
const unsigned
int8 AIR_PA_
TABLE[8]=
{
```


■ PHOTO 3. The A2500R24A-EM1 is the marriage of an A2500R24A AIR module and a Texas Instruments-inspired daughterboard.




■ PHOTO 4. The AIR PLANE eliminates the need to permanently mount an AIR module in the hardware/firmware development phase of the design cycle.

```
        0xFE,0x00,0x00,0x00,0x00,0x00,0x00,0x00
};
```

The transmit FIFO and receive FIFO handle the module's outgoing and incoming data, respectively. The AIR module manipulates the configuration register values and FIFO data using a state machine that runs within its CC2500 core. The host microcontroller can control the movement between states by issuing strobes. Strobes are really commands such as start receiving (SRX) or start transmitting (STX). The AIR module's available command strobes are contained within **Listing 3**. The AIR module's registers, FIFOs, and command strobes are all laid out for us. It's up to us to manipulate these resources in such a way as to cause the transmission and reception of digital data. So, let's get with it.

## AIR TOOLS

We must initiate an AIR module RESET before any register manipulation can take place:

```
//***********************************************
//* RESET AIR
//***********************************************
void reset_air(void)
{
       DISABLE_SPI;
       output_bit(SCLK,1);
       output_bit(MOSI,1);
       output_bit(CSN,1);
       delay_ms(1);
       output_bit(CSN,0);
       delay_ms(1);
       output_bit(CSN,1);
       delay_ms(1);
       output_bit(CSN,0);
       while(input(MISO));
       ENABLE_SPI;
       data_out = AIR_SRES;
       write_data;
       DISABLE_SPI;
       while(input(MISO));
       output_bit(MOSI,0);
       output_bit(SCLK,0);
       output_bit(CSN,1);
       ENABLE_SPI;
}
```

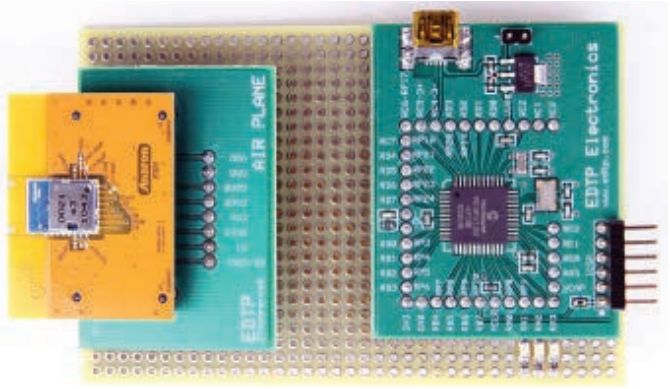
The first command strobe (AIR_SRES) is executed within the AIR module RESET function. Note that the SPI portal is alternately enabled and disabled in the *reset_air* function. The reason for this is that we must wait for MISO to fall logically low before engaging the AIR module via its SPI portal. Following the assertion of the CSn signal by the host microcontroller, the AIR module forces its SO pin logically low to indicate that its crystal oscillator is running. This logic low state on the AIR module's SO pin is identified as the CHIP_RDYn signal. You'll see this wait for CHIP_RDYn sequence often in the AIR support functions we will create. In that there are default values loaded into the AIR module's registers after reset, the logical starting point in our firmware generation process is to write a function to read the AIR module registers:

```
//***********************************************
//* MACROS
//***********************************************
#define NOP           delay_cycles(1)
#define LED_ON(led)   output_bit(led,0)
#define LED_OFF(led)  output_bit(led,1)
#define enable_air    output_bit(CSN,0)
#define disable_air   output_bit(CSN,1)
#define xfer_data     data_in =
spi_read(data_out)
#define read_data     data_in = spi_read(0)
#define write_data    spi_write(data_out)
#define ENABLE_SPI    setup_spi(SPI_MASTER|SPI_
          L_TO_H|SPI_XMIT_L_TO_H|SPI_CLK_DIV_16);
#define DISABLE_SPI   setup_spi(SPI_DISABLED)
//***********************************************
//* READ AIR REGISTER
//***********************************************
void read_air_reg(unsigned int8 baddr)
{
       DISABLE_SPI;
       enable_air;

       while(input(MISO));
       ENABLE_SPI;
       data_out = baddr | 0x80;
       write_data;
       if(spi_data_is_in())
       {
              read_data;
       }
       disable_air;
}
```

I've posted the macro definitions here for clarity. The *read_air_reg* function disables the microcontroller's SPI portal to allow the microcontroller's MISO pin to act as a digital input to detect the CHIP_RDYn signal. Once MISO goes low, the SPI portal is reactivated, the desired register address with the read bit enabled (baddr | 0x80) is transferred to the AIR module, and the contents of the addressed register are returned to the microcontroller. Now that we have a method to read the AIR module registers, we can build a write function:

```
//***********************************************
//* WRITE AIR REGISTER
//***********************************************
void write_air_reg(unsigned int8 baddr,unsigned
int8 bdata)
{
       DISABLE_SPI;
       enable_air;
       while(input(MISO));
       ENABLE_SPI;
       data_out = baddr;
       xfer_data;

       status_byte = data_in;
       data_out = bdata;
       xfer_data;
       disable_air;
}
```

Note that along with writing the data, we simultaneously obtain a status byte from the AIR module. Anytime a header, data byte, or command strobe is sent on the SPI




■ PHOTO 5. The PIC18F47J53 module, the AIR PLANE, and its evaluation board cargo are all fitted on an EDTP plated-through perf board. The electrical connections are made with point-to-point solder techniques and wire wrap.

```
//***********************************************
//* AIR CONFIGURATION REGISTERS
//***********************************************
#define AIR_IOCFG2    0x00   // GDO2 output pin
                             // configuration
#define AIR_IOCFG1    0x01   // GDO1 output pin
                             // configuration
#define AIR_IOCFG0    0x02   // GDO0 output pin
                             // configuration
#define AIR_FIFOTHR   0x03   // RX FIFO and TX
                             // FIFO thresholds
#define AIR_SYNC1     0x04   // Sync word, high
                             // byte
#define AIR_SYNC0     0x05   // Sync word, low
                             // byte
#define AIR_PKTLEN    0x06   // Packet length
#define AIR_PKTCTRL1  0x07   // Packet automation
                             // control
#define AIR_PKTCTRL0  0x08   // Packet automation
                             // control
#define AIR_ADDR      0x09   // Device address
#define AIR_CHANNR    0x0A   // Channel number
#define AIR_FSCTRL1   0x0B   // Frequency
                           // synthesizer control
#define AIR_FSCTRL0   0x0C   // Frequency
                           // synthesizer control
#define AIR_FREQ2     0x0D   // Frequency control
                             // word, high byte
#define AIR_FREQ1     0x0E   // Frequency control
                             // word, middle byte
#define AIR_FREQ0     0x0F   // Frequency control
                             // word, low byte
#define AIR_MDMCFG4   0x10   // Modem
                             // configuration
#define AIR_MDMCFG3   0x11   // Modem
                             // configuration
#define AIR_MDMCFG2   0x12   // Modem
                             // configuration
#define AIR_MDMCFG1   0x13   // Modem
                             // configuration
#define AIR_MDMCFG0   0x14   // Modem
                             // configuration
#define AIR_DEVIATN   0x15   // Modem deviation
                             // setting
#define AIR_MCSM2     0x16   // Main Radio Cntrl
                         // State Machine config
#define AIR_MCSM1     0x17   // Main Radio Cntrl
                         // State Machine config
#define AIR_MCSM0     0x18   // Main Radio Cntrl
                         // State Machine config
#define AIR_FOCCFG    0x19   // Frequency Offset
                         // Compensation config
#define AIR_BSCFG     0x1A   // Bit
                   // Synchronization configuration
#define AIR_AGCCTRL2  0x1B   // AGC control
#define AIR_AGCCTRL1  0x1C   // AGC control
#define AIR_AGCCTRL0  0x1D   // AGC control
#define AIR_WOREVT1   0x1E   // High byte Event 0
                             // timeout
#define AIR_WOREVT0   0x1F   // Low byte Event 0
                             // timeout
#define AIR_WORCTRL   0x20   // Wake On Radio
                             // control
#define AIR_FREND1    0x21   // Front end RX
                             // configuration
#define AIR_FREND0    0x22   // Front end TX
                             // configuration
#define AIR_FSCAL3    0x23   // Frequency
                           // synthesizer calibration
#define AIR_FSCAL2    0x24   // Frequency
                           // synthesizer calibration
#define AIR_FSCAL1    0x25   // Frequency
                           // synthesizer calibration
#define AIR_FSCAL0    0x26   // Frequency
                           // synthesizer calibration
#define AIR_RCCTRL1   0x27   // RC oscillator
                             // configuration
#define AIR_RCCTRL0   0x28   // RC oscillator
                             // configuration
#define AIR_FSTEST    0x29   // Frequency
                           // synthesizer cal control
#define AIR_PTEST     0x2A   // Production test
#define AIR_AGCTEST   0x2B   // AGC test
#define AIR_TEST2     0x2C   // Various test
                             // settings
#define AIR_TEST1     0x2D   // Various test
                             // settings
#define AIR_TEST0     0x2E   // Various test
                             // settings
```

■ **LISTING 1.** These 47 registers can be considered as the AIR module's knobs. All of the AIR module's RF and data handling parameters are controlled by the values within these registers.

portal a status byte is returned by the AIR module on it's SO line. The layout of the status byte is laid out in **Figure 1**. Let's try out our new *write_air_reg* function:

```
init();
write_air_reg(AIR_PTEST,0x7F);
```

The *init* function has configured the PIC18F47J53 hardware, reset the AIR module, loaded the AIR module configuration registers and PATABLE, and placed the AIR module in the IDLE state. The transmit and receive FIFOs are also cleared during the initialization. The write to the AIR_PTEST register should return a status byte informing us that the CHIP_RDYn signal is logically low, the current AIR module state is IDLE, and there are more than 15 bytes free in the FIFOs. **Screenshot 1** captures the contents of an MPLAB Watch window that contains the returned status byte value. Using **Figure 1** to decode the value of the status byte verifies my predictions. We need not change the value of a register or issue a meaningful command strobe to obtain a status byte. We can also trigger the issuance of a status byte by issuing a NOP (No Operation) command strobe:

```
init();
send_strobe(AIR_SNOP);
```

Sending command strobes is also essential to getting on the AIR. Just like the read and write register functions, there's no rocket science behind the *send_strobe* function. Here's what the send_strobe function code looks like:

```
//***********************************************
//* SEND AIR STROBE
//***********************************************
void send_strobe(unsigned int8 bstrobe)
{
        DISABLE_SPI;
        enable_air;

        while(input(MISO));
        ENABLE_SPI;
        data_out = bstrobe;
        xfer_data;
        status_byte = data_in;
        disable_air;
}
```

The *xfer_data* macro — which gleans a status byte — is based on a built-in SPI function of the CCS C compiler.

## AIR TRANSIT

I think we're ready to fly some bits around. Let's code up a transmitter. Here's what needs to happen:

```
#ifdef TRANSMITTER
              if(++usbcntr > 40000)
              {
                     AIR_idle_mode;
                     AIR_clear_tx_fifo;
                     build_tx_pkt(10,0x22);
                     send_pkt(payloadlen + 1);
```

```
//***********************************************
//* AIR STATUS REGISTERS
//***********************************************
#define AIR_PARTNUM   0x30    // Part number
#define AIR_VERSION   0x31    // Current version
                              // number
#define AIR_FREQEST   0x32    // Frequency offset
                              // estimate
#define AIR_LQI       0x33    // Demodulator
                             // estimate for link quality
#define AIR_RSSI      0x34    // Received signal
                             // strength indication
#define AIR_MARCSTATE0x35     // Control state
                              // machine state
#define AIR_WORTIME1  0x36    // High byte of WOR
                              // timer
#define AIR_WORTIME0  0x37    // Low byte of WOR
                              // timer
#define AIR_PKTSTATUS0x38     // Current GDOx
                       // status and packet status
#define AIR_VCO_VC_DAC       0x39    // Current
                             // setting from PLL cal module
#define AIR_TXBYTES   0x3A    // Underflow and #
                              // of bytes in TXFIFO
#define AIR_RXBYTES   0x3B    // Overflow and # of
                              // bytes in RXFIFO
//***********************************************
//* AIR PATABLE REGISTER
//***********************************************
#define PATABLE       0x3E
//***********************************************
//* AIR FIFO REGISTER
//***********************************************
#define TXFIFO        0x3F
#define RXFIFO        0x3F
```

■ **LISTING 2.** The AIR module status registers contain information on everything from the version of the core to the number of bytes in the receive and transmit queues. Oh yeah, the signal strength (RSSI) can also be found in the status register area.

```
                    output_toggle(BLED);
                    usbcntr = 0;
                }
#endif
```

The *init* function is identical for both the transmitter and receiver, and will not complete until the PIC18F47J53 is enumerated and goes online with the PC's USB portal. The CCS C compiler's *usbtask* function must be called periodically; the *usbtask* function call is included in the endless *do loop* that makes up the *main* function. I arbitrarily chose the name *usbcntr* for the 16-bit memory location that holds the number of cycles through the transmitter routine. The *usbcntr* value determines when to blink the blue LED. The self-explanatory AIR_idle_mode and AIR_clear_tx_fifo are command strobes in the form of macros. With the AIR module idling and the transmit FIFO cleared, we have indicated that we want to build a packet that is 10 bytes in length and send it to a receiver with the address of 0x22. To do this, some ground work must be laid first. In the *init* function, we loaded the AIR module's configuration registers with a modified template of values obtained from the Anaren website. The original set of AIR module configuration values called *original-config-values.h* is part of the article download package. If you compare the original set of configuration values with our modified configuration values in *air_rf_settings.h*, you'll see that we modified the AIR_PKTCTRL1 register. AIR_PKTCTRL1's original value was 0x04 which appends the RSSI and LQI bytes to the end of our packet. Changing the AIR_PKTCTRL1 value to 0x07 adds a packet address check to the packet's appended RSSI and LQI bytes. Now that address checking is in effect, we need to specify a receiver address in the AIR_ADDR configuration register. According to our *build_tx_pkt* function, the receiver's address should be 0x22 and that is reflected in the receiver's *air_rf_settings.h* file. Let's flesh out the *build_tx_pkt* function:

```
//***********************************************
//* BUILD TX PACKET
//***********************************************
void build_tx_pkt(unsigned int8 len,unsigned
int8 addr)
{
       unsigned int8 i;
       payloadlen = len;

       tx_buf[0] = payloadlen;
       tx_buf[1] = addr;
       for(i=2;i<payloadlen+1;++i)
       {

              tx_buf[i] = i;
       }
}
```

In variable length packet mode (AIR_PKCTRL least significant 2 bits = 01), the length of the packet is the first byte in the packet, and the address byte is next followed by the data. Once the packet is assembled, we can send it:

```
//***********************************************
//* SEND PACKET MANUAL
```

```
//***********************************************
//* AIR STROBES
//***********************************************
#define AIR_SRES      0x30    // Reset chip.
#define AIR_SFSTXON   0x31    // Enable and
       // calibrate frequency synthesizer (if
       // MCSM0.FS_AUTOCAL=1).
       // If in RX/TX: Go to a wait state where
       // only the synthesizer is running (for
       // quick RX / TX turnaround).
#define AIR_SXOFF     0x32    // Turn off crystal
                              // oscillator.
#define AIR_SCAL      0x33    // Calibrate
       // frequency synthesizer and turn it
       // off(enables quick start).
#define AIR_SRX       0x34    // Enable RX.
       // Perform calibration first if coming
       // from IDLE and MCSM0.FS_AUTOCAL=1.
#define AIR_STX       0x35    // In IDLE state:
       // Enable TX. Perform calibration first
       // if MCSM0.FS_AUTOCAL=1. If in RX state
       // and CCA is enabled: Only go to TX if
       // channel is clear.
#define AIR_SIDLE     0x36    // Exit RX / TX,
       // turn off frequency synthesizer and
       // exit Wake-On-Radio mode if applicable.
#define AIR_SAFC      0x37    // Perform AFC
       // adjustment of the frequency
       // synthesizer
#define AIR_SWOR      0x38    // Start automatic
       // RX polling sequence (Wake-on-Radio)
#define AIR_SPWD      0x39    // Enter power down
       // mode when CSn goes high.
#define AIR_SFRX      0x3A    // Flush the RX FIFO
                              // buffer.
#define AIR_SFTX      0x3B    // Flush the TX FIFO
                              // buffer.
#define AIR_SWORRST   0x3C    // Reset real time
                              // clock.
#define AIR_SNOP      0x3D    // No operation. May
       // be used to pad strobe commands to two
       // bytes for simpler software.
```

■ **LISTING 3.** Command strobes are issued by the host microcontroller to traverse the AIR module's internal state machine.

| Bits | Name | Description |
|---|---|---|
| 7 | CHIP_RDYn | Stays high until power and crystal have stabilized. Should always be low when using the SPI interface. |
| 6:4 | STATE[2:0] | Indicates the current main state machine mode |

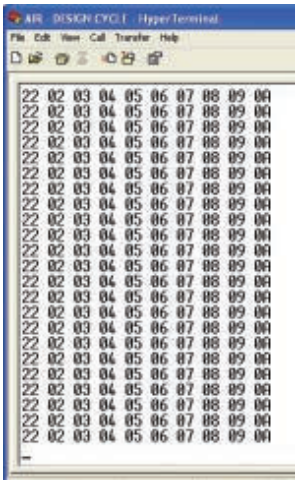
| Value | State | Description |
|---|---|---|
| 000 | IDLE | Idle state (Also reported for some transitional states instead of SETTLING or CALIBRATE) |
| 001 | RX | Receive mode |
| 010 | TX | Transmit mode |
| 011 | FSTXON | Frequency synthesizer is on, ready to start transmitting |
| 100 | CALIBRATE | Frequency synthesizer calibration is running |
| 101 | SETTLING | PLL is settling |
| 110 | RXFIFO_OVERFLOW | RX FIFO has overflowed. Read out any useful data, then flush the FIFO with SFRX |
| 111 | TXFIFO_UNDERFLOW | TX FIFO has underflowed. Acknowledge with SFTX |

| | | |
|---|---|---|
| 3:0 | FIFO_BYTES_AVAILABLE[3:0] | The number of bytes available in the RX FIFO or free bytes in the TX FIFO |

■ FIGURE 1. The data contained within the status byte comes in handy when you need to quickly assess the status of the state machine and FIFOs.

```
//***********************************************
void send_pkt(unsigned int8 pktsize)
{
        unsigned int8 i;

        DISABLE_SPI;
        enable_air;
        while(input(MISO));
        ENABLE_SPI;
        data_out = TXFIFO + 0x40;
        write_data;
        for(i=0;i<pktsize;i++)
        {
                spi_write(tx_buf[i]);

        }
        disable_air;
        AIR_transmit_mode;
        while(input(GDO0)==0);
        while(input(GDO0));
        AIR_idle_mode;
}
```

The 0x40 added to the TXFIFO address allows us to burst write to the transmit FIFO. Burst writing/reading allows us to drop the CSn signal logically low and continually stream data until we raise the CSn signal which signals the end of the data burst. We can apply the same TXFIFO bursting logic to RXFIFO bursting. Take a look at the code that makes up a bursting RXFIFO read:




■ SCREENSHOT 2. A 10-byte packet delivered as ordered to the receiver at address 0x22. What you don't see here is the length byte we read and didn't write to the receive buffer, and a CRC at the end of the packet.

```
//************************
//* READ RXFIFO
//************************

void read_rxfifo(unsigned
int8 *buf,unsigned int8
len)
{
unsigned int i;
DISABLE_SPI;
enable_air;

while(input(MISO));
ENABLE_SPI;
data_out = RXFIFO | 0xC0;
write_data;
for(i = 0;i < len;i++)
{
        buf[i] = spi_read(0);
}
disable_air;
}
```

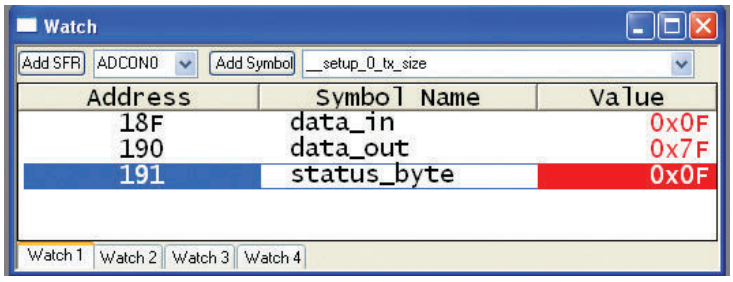
The 0xC0 that is ORed to the RXFIFO address tells the module to burst the read operation. Data is streamed from the RXFIFO to the PIC18F47J53 until the CSn line is returned logically high. Here's my idea of how to receive data sent by our TRANSMITTER code:

```
#ifdef RECEIVER
        AIR_receive_mode;     //enter receive mode
        while(input(GDO0)==0);
        while(input(GDO0));
        AIR_idle_mode;        //packet received

        read_air_reg(RXFIFO);//get length byte
        rx_len = data_in;
        read_rxfifo(rx_buf,rx_len);
        //read address byte and data
        read_rxfifo(rf_stats,2);
        //read LQI and RSSI
        for(i=0;i<rx_len;i++)
        {
           printf(usb_cdc_putc, "%X
           \r\n",rx_buf[i]);
        }
        output_toggle(BLED);
#endif
```

After entering receive mode and receiving a packet — which is signaled by the toggling of the AIR module's GDO0 I/O pin — we place the AIR module in IDLE mode and read the length byte of the received packet. The amount of data specified by the length byte is burst read into the receive buffer rx_buf. The appended RSSI and LQI bytes are bursted into the rf_stats array. If all went as planned, we should be able to transmit the contents of the receive buffer to a HyperTerminal session via the PIC18F47J53's USB portal.

## A BREATH OF FRESH AIR

I'll leave you with **Screenshot 2**. The AIR Tools source code is included in its entirety within the download package that accompanies this edition of Design Cycle. I'll also include the AIR PLANE ExpressPCB layout file for those of you that want to scratch-build your own AIR craft. You're allowed to wear that pointy hat adorned with stars and moons as AIR is now in your Design Cycle. **NV**




■ SCREENSHOT 1. A read operation returns the number of free bytes in the receive FIFO. Conversely, a write operation returns the number of free bytes in the transmit FIFO. When the FIFO_BYTES_AVAILABLE is equal to 15, 15 or more bytes are available.

Anaren
AIR A2500R24x Modules
AIR A2500R24A-EM1
**www.anaren.com**

Custom Computer Services
CCS C Compiler
**www.ccsinfo.com**

Microchip
PIC18F47J53 Microcontroller
**www.microchip.com**