

Supplementary Information on Programming for Your Radio – Icom

- (1) Go to the Radios and controllers section of the main code. You will need to set the CIV address of your transceiver in the mainrig variable (Icom only) as well as the baud rate in the radiobaud variable. If you are not using the Due, this value must be less than 65,535.
- (2) Decide which pushbutton functions you want on your panel. Go to the section called “Constructors for controls and display”. Beneath it is a section called “Pushbuttons that control the radio”. Here you will see five buttons defined. Each button contains three parameters: the first is the Arduino digital pin that is connected to the button, the second is the byte for the CIV command, and the third is the byte for the CIV subcommand. Enter the appropriate values from your radio’s manual for the function you want to control. Beneath the line for each button is a line of code beginning with “char” (to define a character string variable). Replace the words in quotes with the information that you want to appear on the display for that button. The first field called B1Label denotes the text label for button 1. The fields S1Label1 and S1Label2 contain the state labels for that button (in this case, Off and On). Modify these values for each button as appropriate. Then go to the section called CIVPoll and change the bytes for the control you just set up to the same values as was used in the button constructor. These lines enable the controller to retrieve the state for that button or dial from the radio should you make an adjustment there and not on the controller.
- (3) Decide which dial functions you want on your panel. They will need to be defined in the Dials section of the code similarly to what is described in (2) above for pushbuttons. Note that encoder dials have two digital pins – one for detecting movement in either direction. You can swap the two pin values to change the sense of rotation on the dial. The only text field that you need to adjust for dials is the label, which appears on the line beneath that dial’s constructor. Once you set up a dial, go to the CIVPoll constructor section and modify the command and subcommand bytes for that dial there as well.
- (4) Finally, modify the CIVPoll constructors to pull in the appropriate data from your radio. Replace the HEX command and subcommand bytes with the appropriate values for your controls

This process will set you up with a controller having six buttons and six encoder dials. If you want a different number of controls, you will need to add or delete constructors and also modify other sections of the program.

The other program that is included is DialBoxIcomKE1Uversion. This program has a few different options for buttons. As an example, my Icom transceiver has manual notch and automatic notch functions that are controlled with the same menu option on the radio. I decided to implement that control as a tri-state switch, wherein pressing the same button toggles between no notch, manual notch and auto notch options. There are separate Icom CIV commands for manual notch and autonotch. The logic is explained in the commented code, and you can use that section as a model if you want tri-state switches on your panel. Likewise, there are a few buttons in that implementation that change controller parameters but do not directly communicate to the radio. For example, there are two buttons that increment or decrement the memory counter for the voice keyer in the radio. While they do not change the setting in the radio, they influence which memory send command is sent to the radio when the button for the voice keyer is pressed on the controller.

Description of how the Icom library works:

Updating the status of buttons:

```
bool Button::update() // Update function for a button used as above.
{
    if (digitalRead(pinnumber)==LOW) {
        delay(200); // (eliminate button bounce)
        return true; // Return true if pressed.
    }
    else return false; // Return false if pressed.
}
```

The function above senses whether a button was pressed. If it was pressed, it returns a value of “true”, which causes the main code to invert the level of that button from 0 to 1 (or vice versa). That prompts the program to call the Senddata function for that button.

```
void Button::Senddata(byte radio, byte ctrlr) // Function to send data to the radio when a button is
pressed
{
    for (int i=0; i<2; i++) {
        Serial1.write (header[i]);
    }
    Serial1.write (radio);
    Serial1.write (ctrlr);
    Serial1.write (command);
    Serial1.write (subcommand);
    Serial1.write (level);
    Serial1.write (0xFD);
    Serial1.flush(); // wait until all data sent
    delay(10);
}
```

In the first part of the program, you defined bytes for the radio and the controller. Those need to be sent as part of the byte string for the radio, so they are imported into this on the first line (byte radio, byte ctrlr).

We next send the header of FE FE:

```
for (int i=0; i<2; i++) {
    Serial1.write (header[i]);
}
```

Finally, we send the actual data as a series of bytes in the sequence: radio ID, controller ID, Command, Subcommand, Data (which is represented as level for the control):

```
Serial1.write (radio);
```

```

Serial1.write (ctrlr);
Serial1.write (command);
Serial1.write (subcommand);
Serial1.write (level);

```

Followed by the “FD” footer. Note all hex values are denoted as such by the designator 0x. Thus, 0xFD means FD as a hexadecimal representation of a byte.

```

Serial1.write (0xFD);
Serial1.flush(); // wait until all data sent
delay(10);

```

Updating the status of dials:

Controls are updated as follows:

```

bool Controls::update(int step) {
thisEncoder.tick();
controldir=thisEncoder.getDirection();
if (step == 1){
if ((level>0 && level <100) || (level == 0 && controldir >0) || (level == 100 & controldir <0)) {
level += controldir*step;
}
}
if (step == 5){
if ((level>4 && level <96) || (level <5 && controldir >0) || (level >94 && controldir <0)) {
level += controldir*step;
}
}
if (controldir != 0)
{
return true;
}
else return false;
}

```

In the code above, step defines whether this is going to be a fine adjustment or a coarse adjustment. For fine adjustment, the control steps in increments of 1. For a coarse adjustment it is in increments of 5. The increment is set by the Coarse/Fine button on the controller.

The conditional lines check to see that the control is in-bounds (between values of 0 and 100). If it is, then rotating the control to the right adds to its value by the increment, and rotating it left decreases the value by the increment. If such a change would take the control value out of range (<0 or >100), then turning the control has no effect. If the control was moved, then the function returns true, and its level is updated on the display and sent to the radio.

```

void Controls::Senddata(byte radio, byte ctrlr)
{

```

// The following lines send an untranslated decimal number in hex format across two bytes. i.e. 255 is sent as "0255".

```
normlevel=level*256/100;
data[0] = normlevel/100;
data[1] = (normlevel%100/10*16)+normlevel%10;
for (int j=0; j<2; j++) {
    Serial1.write (header[j]);
}
Serial1.write (radio);
Serial1.write (ctrlr);
Serial1.write (command);
Serial1.write (subcommand);
for (int i=0; i<2; i++) {
    Serial1.write (data[i]);
}
Serial1.write (0xFD);
Serial1.flush(); // wait until all data sent
success=true; // delete this line when implementing.
delay(10);
}
```

The function above sends the updated value of a control. All Yaesu and Kenwood data that is sent with a dial is encoded as BCD (binary coded decimal). Therefore, we need to build a set of bytes that represents each digit in its decimal representation. Note that a byte is two hex digits: AB, where A is the 16's place, and B is the ones place. The range of the data will be from 0 to 256, so the first thing we do is scale from 0-100 to 0-256.

```
normlevel=level*256/100;
```

We start by determining the hundreds place (data[0]), which is the integer that results from dividing our control's level by 100. Since we are not doing floating point math, the remainder is dropped. This value will occupy the ones position of our first data byte.

```
data[0] = normlevel/100;
```

We then determine the tens level, which we do by taking the remainder of the hundreds division and divide by 10. Since we are not doing floating point math, the remainder is truncated, and we are left with the tens digit. This occupies the 16's digit of the byte, so we multiply by 16. Finally, we can add the one's place to this, so we take the remainder from the 10's division and just add it.

```
data[1] = (normlevel%100/10*16)+normlevel%10;
```

The subsequent lines send the string of bytes in analogous fashion to what was done for the buttons.

Fetching data from the radio:

This is the most complicated part of the software:

```
bool CIVPoll::UpdateCIV(byte radio, byte ctrlr)
```

This function takes in the value of the radio and the controller as before to construct the interrogation string.

```
    for (int j=0; j<2; j++) {  
        Serial1.write (header[j]);  
    }  
    Serial1.write (radio);  
    Serial1.write (ctrlr);  
    Serial1.write (command);  
    Serial1.write (subcommand);  
    Serial1.write (0xFD);  
    Serial1.flush();
```

This code sends the interrogation string as

FE FE RadioByte ControllerByte CommandByte SubcommandByte FD

```
CAT_TRX_next:  
if (Serial1.available() >0) {  
    incomingTRX = Serial1.read();  
    if ((incomingTRX == 254) ) {  
        goto CAT_TRX_start;  
    }  
}
```

The code above waits for data to come in on Serial1. If it does, it reads the first byte (Serial1.read();) and checks to see if that byte is FE (decimal 254) as we expect it to be (headers from the radio are FE FE). If it is, then we go to CAT_TRX_start.

Otherwise, we go back to the beginning (CAT_TRX_next) and try again. If the number of tries exceeds the retry count, then the data read fails, and the control value is not updated (goto invalid).

The next section pulls in all of the remaining data on the binary feed. There will be at most 9 bytes, so we read character by character 9 times.

```
for (int i = 0; i < 10; i++) {  
    if (Serial1.available() > 0) {  
        buffget_CAT[i] = Serial1.read();
```

We then concatenate the bytes into a string:

```
String byte_string = String(buffget_CAT[i], HEX);
```

And we wait for the footer byte of FD to come in at the end. If it does, we go to the subroutine to process the string.

```
    if (buffget_CAT[i] == 253 ) {
```

```

    goto process_string_TRX;
}

```

If this process fails, we go to the invalid read section and return a value of false. For this description, let's assume the read was good:

```

if ((buffget_CAT[0] == 254)) {
    retrycount=0;
    goto valid_string_TRX;
}

```

The code above checks that the second byte is also FE (the second byte of the header). If so, we go to valid_string_TRX and reset the retry counter.

```

if (buffget_CAT[3] == command && buffget_CAT[4] == subcommand) {

```

The first line of that routine (above) checks to be sure that the 4th byte and the 5th byte match (indexes for arrays always start at 0) match the command and the subcommand that we are interrogating. If so, then we process the rest of the string.

```

    if (command == 0x16) { // Get a 0 or 1 for all of our buttons.
        level = buffget_CAT[5]; // will be 0 or 1 if off or on.
        return true;
    }

```

These lines check to see if the command is 16. If so, then we are getting data for a button. **Note, all of my buttons use command 16. If you have buttons that use other commands, then you will need to copy this code and paste it immediately below for each additional command you have that uses buttons. Just copy/paste and change 0x16 to 0xYY where YY equals the byte for your other button commands.**

```

    if (command == 0x14) { // Get the number (0 to 255) for all of our controls.
        level = (buffget_CAT[5]&0x0F) * 100; // gets the 100s digit of the BCD data.
        level = level + (((buffget_CAT[6]>>4)*10)+(buffget_CAT[6]&0x0F)); // Gets the 10s and 1s digits and
        adds them. We should now have 0 <=level<256;
        rawlevel = level*100/256; // Normalize this to 0-100, which is the range for all of our controls.
        level = rawlevel;
        if (level < 101) {
            return true; // go back to the main page.
        }
    }
    else return false; // data is out of range, so do not update the display.
}

```

This set of lines checks to see if the command is 14. **As above, all of my dials use command 14. If you have dials with other command groups, then replicate this code and replace 0x14 with 0xYY, where YY is the byte for your other dial commands.**

```

    level = (buffget_CAT[5]&0x0F) * 100;

```

```
level = level + (((buffget_CAT[6]>>4)*10)+(buffget_CAT[6]&0x0F));
```

These two lines convert the BCD values coming in to a decimal value called level. It essentially does the reverse of what was described above for sending data.

```
rawlevel = level*100/256; // Normalize this to 0-100, which is the range for all of our controls.  
level = rawlevel;  
if (level < 101) {  
    return true; // go back to the main page.
```

These lines convert the data from a range of 0-256 to a range of 0-100. If the data are in range, we return a value of true, which goes back to the main program, updates the actual level of the control (level) and updates the display.

Otherwise, we have a false read, and the actual level of the control is not changed. The program moves on.