

Supplementary Information on Programming for Your Radio – Yaesu and Kenwood

Note – the author does not have a Yaesu or Kenwood radio, so the validity of this code could not be checked other than using a terminal monitor to validate the structure of the text going out on the serial port. You may need to modify this code to get it to work properly!

- (1) Go to the Radios and controllers section of the main code. You will need to set 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 a set of const char* arrays that contain three parameters. The first is the two-character radio command for the button, and the subsequent fields define the possible states for that button.

Beneath the line for each button constructor 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.

- (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 text field after the pin definitions corresponds to the radio command for that dial. The subsequent numbers define the minimum and maximum values for that dial.
- (4) Finally, modify the CIVPoll constructors to pull in the appropriate data from your radio. Replace the text command in quotes with the appropriate value for your control. The following two values correspond to the number of data characters expected and the maximum value of the control. These values are used to parse and check validity of the received data.

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. While this program is set up for Icom radios, it serves as an example of how to modify some of the code in the prior sections to handle different types of controls. As an example, my 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 Yaesu 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() // Function to send data to the radio when a button is pressed
{
    Serial.print(_command[0]); // the first member of the array is the Yaesu command name
    Serial.print(_command[level+1]); // the other members of the array are indexed by level (0 or 1 for off
or on). You can have more states if needed.
    Serial.println(";"); // semicolon is the Yaesu terminator for all command lines.
    Serial.flush(); // wait until all data sent
    delay(10);
}
```

In the first part of the program, you defined a character array that corresponded to the command structure for a given button (command, state1, state2, etc.). For example:

```
const char* B1cmd[] = {"BI", "0", "1"};
```

In this case, the command is BI, and it has two states: 0 and 1. The array has three members: BI, 0 and 1. They have index values of 0, 1 and 2, respectively. If you add more states, they will have index values that increase by 1 for each additional state that is added. For a two-state button, its variable “level” will be either 0 or 1 as above. The string to send to the radio is constructed on the two Serial.print lines. The first thing that is sent is the command BI (which is index 0 in the array above).

```
Serial.print(_command[0]); // the first member of the array is the Yaesu command name
```

The second thing that is sent is the state, which either has index 1 or 2 in the array above. Thus, we take “level”, add 1 to it to correspond to the index of the two states (B1cmd[1] = 0 and B1cmd[2] = 1).

```
Serial.print(_command[level+1]);
```

Finally, a semicolon is sent as the terminator (with a linefeed).

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()
{
// The following lines normalize the control level to what the radio is expecting and sends it as a three
digit number with leading zeroes.
normlevel=level*(_high - _low)/100;
Serial.print (_command);
if (normlevel<100) Serial.print ("0");
if (normlevel<10) Serial.print ("0");
Serial.print (normlevel);
Serial.println(";"); // Semicolon is the terminator for all Yaesu lines.
```

```

Serial.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 has three characters (i.e. 010 for 10, 005 for 5, 100 for 100, etc.). Therefore, we need to build a string with leading zeroes if needed.

While all of the control data is relayed in levels of 0-100, commands have different ranges of valid data. Therefore, we have to normalize the percent max. value of the control (level) to what the radio expects. This is done using the command:

```
normlevel=level*(_high - _low)/100;
```

Next, we send the two character command that is associated with the control (from the input you provided in the dial constructor):

```
Serial.print (_command);
```

And finally, we send the data with leading zeroes:

```

if (normlevel<100) Serial.print ("0");
if (normlevel<10) Serial.print ("0");
Serial.print (normlevel);

```

Followed by the semicolon terminus.

Fetching data from the radio:

This is the most complicated part of the software:

```
CIVPoll::UpdateCIV()
```

This function begins by printing the function that will be interrogated. The main program polls every 5 seconds.

```
Serial.println (_command);
```

We then expect the radio to send a line that begins with that same command. Therefore, we will read the first characters that come back and make sure they match. Because commands have different lengths, we identify the length of this header:

```
length = strlen(_command)-1;
```

Next, we wait for data to come in on the serial port, and when it does, we parse the text:

```
if (Serial.available() >0) {
```

```

Serial.readBytes(incomingTRX,length);
for (int i=0; i<=length; i++) {
  if (incomingTRX[i] == _command[i]) { // We are testing to see if every character received matches the
    command
    j++; // If so, increment the matching character counter
  }
}

```

The above block of code brings in only the first n characters (where n is the length of the command) and looks at them character by character to see that they match the command that was sent. The variable j is a counter that keeps track of how many of these characters match. They should all match, and if they do, then we have a valid reply from the radio.

```

if (j==length) {      // If the number of matching characters equals the command length then we have
valid data.
  goto CAT_TRX_start;
}

```

If the reply is valid, we go to CAT_TRX_start. If not, 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 examines the code if the command was read properly.

```

if (Serial.available() > 0) {
  length=Serial.readBytesUntil('; ',buffget_CAT,8);
  _level = 0;
}

```

The section above waits for serial data and then reads what comes in until the semicolon terminator is reached **or** 8 characters are received (the longest data frame possible – modify if you expect longer frames).

We reset the level of the control to zero in that instance.

```

for (int i=1; i<=len; i++) {
  number = (int) buffget_CAT[len-i]; //convert the digits from char to int starting from rightmost
character. The ; terminator was dropped
  number = number-48; // Convert from ASCII to the actual number
  _level = _level + number * pow(10,i-1); //adds the value to level. The pow function converts the
integer to its proper place in the base 10 number.
}

```

This section counts from 1 to the length of the data string. For these radios, the data are always at the end of the string, so we read it from right to left and convert the character at each position to a numerical digit called number. This numerical digit will be the ASCII character code for the number, so we need to subtract 48 from it to convert it to the actual number.

The final step sums the digit to the value “_level” for the control. The first digit we read (place i=1) is the ones digit, followed by the tens digit (i=2), followed by the hundreds digit (i=3). We convert to the

appropriate power of 10 and add to the value of `_level`. Now `_level` represents the true numerical value of the control as an integer.

```
if (_level>0 && _level <= maxlevel) {  
    level = _level;  
    return true;  
}
```

This line checks to be sure that the control's preliminary level (`_level`) is in range. This is why you need to specify the range of the control in the main code. If it is in range, then we have a valid reading. We update the actual level of the control (`level`) and return true to the program. This action will update the display.

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