KX3 Band Decoder/Encoder Theory Of Operations

Introduction

The KX3 Band Decoder/Encoder (KX3 BD/E) is a self contained device that samples data from the KX3 serial port (ACC1), listens for band information and transforms that information into a K3 style ACC connector (Yaesu compatible) band data output. This 15 pin connector is compatible with any K3 compatible band selector device such as the KAT500/KPA500, automatic antenna selectors, automatic tuners, automatic band pass filters and other band aware devices.

Overview

This Theory of Operations is meant to educate the radio operator on how the device actually works. The radio operator is encouraged to look at the schematic and for those interested in software, to review the code (The code makes heavy use of a real time operating system and may be confusing).

Theory of Operation

Hardware TOO

The hardware is in 4 part:

- The Serial port sampling circuit.
- The band data driver circuit.
- The Arduino micro controller.
- The power supply regulator.

The serial port sampling circuit

The KX3 serial port is a modified RS232 signal level serial in and serial out design. A level translator (2n3904) with diode protection is used make use of these signal lines and to avoid changing or interfering with the signals (we are only listening). After the signal is level translated, it is fed into the RXI of the Arduino. There are two level translators. One for the KX3 transmissions and one for the PC transmissions. Only one, the KX3 transmission side, is populated on the board. There is no functionality available for the PC transmissions.

20150531 copyright PDK, Paula Keezer NX1P

The band data driver circuit

Band data is provided by an 8 bit serial shift register. The TPIC6B595 is the same shift register/driver used in Elecraft and other manufacturers equipment. This device was chosen to ensure compatibility with all band selection devices that one would normally connect to the K3. There are 2.2K pull up resistors (to VCC) and .1uf capacitors to ground. This ensures that a floating pin will be observed as a 'high' and that any rf fed back into the band data output will be sent to ground.

The four band selector pins are used with the same band encoding as Elecraft and Yaesu. The schematic also shows 4 additional shift register/driver pins. 2 connected to the 15 pin connector and 2 (plus ground) that are connected to a 3 pin header. These addition connections are non-functional (see final notes).

The Arduino micro controller

A Pro Micro style Arduino micro controller was chosen for this project. This microcontroller serial ports that are independent of the USB developers port (port used to load new code and communicate between the developers PC and the Arduino). Although slightly more expensive, it makes for loading the Arduino code simple.

Not all ports are used by this device. The main ports are 'P1/RX1', P2,P3,P4 for loading/sending and latching the shift register, P5,P6 for serial port switching (not implemented), P7 for mode selection (not implemented) and P8,P9 for led indicators (not implemented).

This design provides regulated 5 volts directly to the Pro Micro. The Pro Micro does have a 12 volt power supply input, however, it is a linear regulator and cannot handle the typical 13.5 volts available in a ham radio application. The 5 volts is provided by a separate regulator circuit.

The power supply regulator

The power supply circuit is made up of a 1.5 amp switching regulator that can switch from 8-19 volts down to 5 volts. The regulator is isolated from the main power input by a 250uh choke and a 1uf tantalum (on the regulator side of choke). This keeps any RF generated by the switch from getting out and effectively keeps any RF picked up by the power supply leads from getting in.

The output of the regulator is fed to the main board through a 1N4005 diode. This is necessary to avoid back feeding the regulator when the Arduino is being powered by the USB port (during program loading and developing, not during normal operations).

Software TOO

The Arduino 'sketch' is compatible with the 1.0.5 development environment. The board selection is the 'Pro Micro 5 volt' chip.   The development environment must be modified to support the ChibOS real time operating system (beyond the scope of this document).

20150531 copyright PDK, Paula Keezer NX1P

The KX3 BD/E sketch makes use of 4 threads (Thread0 throught Thread3), an initialization section and an operating sytem main thread.

Thread0

Thread0 is the serial port sniffer thread. It sleeps for 1ms (a real time OS moment), checks for serial port buffer data, If data then place in a que, evalutate que for band info, construct band info if available and set bandflag, go back to sleep, rinse and repeat.

Thread1

Thread1 checks for bandflag. If bandflag is set, then take band info and send it to the shift register. Go back to sleep.

Thread2

Thread2 simply blinks the on board led's (available on the Pro Micro). They are only visible when the cover is off the KX3 BD/E

Thread3

Thread3 is a diagnostic thread. If the debug port is enabled (the USB port and a TTY serial program connected to it) data will be sent to the port once per second. (review the thread for info about the data being sent). No data will be sent to an inactive USB port.

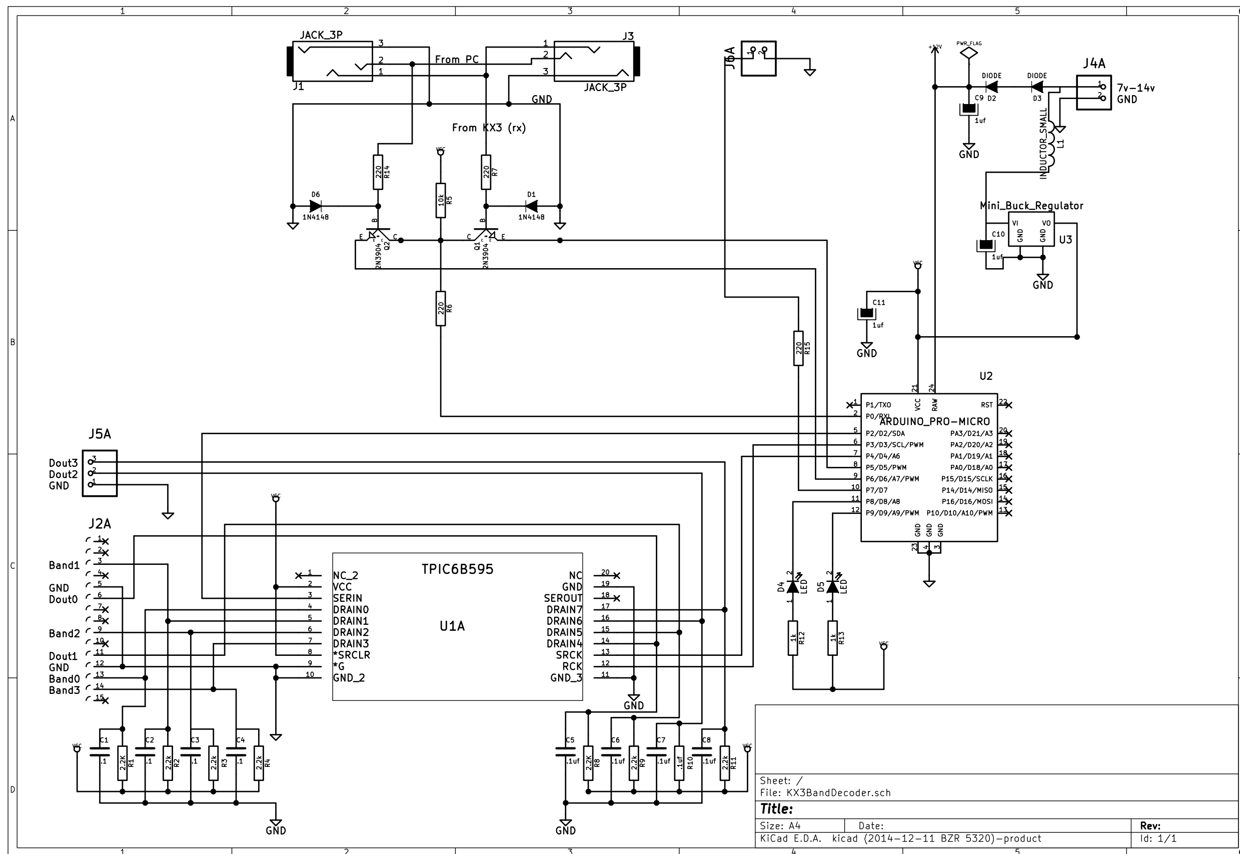
Main Thread

Nothing interesting here. Prior to main thread start up, all other threads are initialized.

20150531 copyright PDK, Paula Keezer NX1P

Appendix

Schematic



Software Source Code

```
// KX3 BD/E
// Dependent on ChibiOS real time OS
// Dependent on Pro Mirco

#include <ChibiOS_AVR.h>

const uint8_t LED_PIN = 17;

const uint8_t SOUT = 2;
const uint8_t SCLK = 4;
const uint8_t RCLK = 3;
const uint8_t SEN1 = 5;
const uint8_t SEN2 = 6;
```

20150531 copyright PDK, Paula Keezer NX1P

```
volatile uint32_t count = 0;

unsigned char bandData = 0;
boolean bandFlag = false;
boolean debug = true;

struct band {
  long int low;
  long int high;
  unsigned char b;
} bands[] = {
  {1800,2000,0x01}
  ,{3500,4000,0x02}
  ,{5000,5100,0x00}
  ,{7000,7300,0x03}
  ,{10100,10150,0x04}
  ,{14000,14350,0x05}
  ,{18068,18168,0x06}
  ,{21000,21450,0x07}
  ,{24890,24990,0x08}
  ,{28000,29700,0x09}
  ,{50000,54000,0x0A}
};


//------------------------------------------------------------
// Thread 0 Serial sniffer
//

#define MAXBUFCNT 80


static WORKING_AREA(waThread0,256);

static msg_t Thread0(void *arg) {

  char temp[10] = "";
  char buf[MAXBUFCNT] = "";
  int bufIdx = 0;

  int i;
  pinMode(SEN1, OUTPUT);
  pinMode(SEN2, OUTPUT);
  digitalWrite(SEN1,LOW);
  digitalWrite(SEN1,HIGH);
```

20150531 copyright PDK, Paula Keezer NX1P

```
  while (1) {
    chThdSleepMilliseconds(1);
    while (Serial1.available()) {
      char c = (char)Serial1.read();
      buf[bufIdx] = c;
      bufIdx++;
      if (bufIdx > MAXBUFCNT) {
        bufIdx--;
      }
      if (c == ';') {
       buf[bufIdx] = '\0';
       if (((buf[0] == 'F') && (buf[1] == 'A')) ||
           ((buf[0] == 'I') && (buf[1] == 'F'))) {
          char *p = &(buf[2]);
          buf[10] = '\0';
          long int freq = atol(p);
          i = 0;
          while (!bandFlag) {
            if ((freq >= bands[i].low) && (freq < bands[i].high)) {
              bandData = bands[i].b;
              bandFlag = true;
              itoa(bandData,temp,10);
              if (debug)
               Serial.println(temp);
            }
            i++;
            if (i > 11) {
              break;
            }
          }
          if (debug) Serial.println(buf);
          char freqstr[16] = "";
          ltoa(freq,freqstr,10);
          if (debug) Serial.println(freqstr);
        }
        bufIdx = 0;
        buf[bufIdx] = '\0';
      }
    }
  }
}

//----------------------------------------------------------
// thread 1 - shift register controller
//
```

20150531 copyright PDK, Paula Keezer NX1P

```
static WORKING_AREA(waThread1, 64);

static msg_t Thread1(void *arg) {
  int i;
  pinMode(SOUT, OUTPUT);
  pinMode(SCLK, OUTPUT);
  pinMode(RCLK, OUTPUT);
  digitalWrite(RCLK,LOW);
  digitalWrite(SCLK,LOW);
  while (1) {
    if (bandFlag) {
      for (i=8;i--;i!=0) {
        if (bandData & 0x80) {
          digitalWrite(SOUT,LOW);
        }
        else {
          digitalWrite(SOUT,HIGH);
        }
        chThdSleepMilliseconds(2);
        digitalWrite(SCLK, HIGH);
        chThdSleepMilliseconds(2);
        digitalWrite(SCLK, LOW);
        chThdSleepMilliseconds(2);

        bandData = bandData << 1;
      }
      chThdSleepMilliseconds(2);
      digitalWrite(RCLK,HIGH);
      chThdSleepMilliseconds(2);
      digitalWrite(RCLK,LOW);
      bandFlag = false;
    }
    chThdSleepMilliseconds(100);
  }
}

//------------------------------------------------------------------------------
// thread 2 - high priority for blinking LED
// 64 byte stack beyond task switch and interrupt needs
static WORKING_AREA(waThread2, 64);

static msg_t Thread2(void *arg) {
  pinMode(LED_PIN, OUTPUT);

  // Flash led every 200 ms.
  while (1) {
```

```
    // Turn LED on.
    digitalWrite(LED_PIN, HIGH);

    // Sleep for 50 milliseconds.
    chThdSleepMilliseconds(50);

    // Turn LED off.
    digitalWrite(LED_PIN, LOW);

    // Sleep for 150 milliseconds.
    chThdSleepMilliseconds(500);
  }
  return 0;
}
//------------------------------------------------------------------------------
// thread 3 - print main thread count every second
// 100 byte stack beyond task switch and interrupt needs
static WORKING_AREA(waThread3, 100);

static msg_t Thread3(void *arg) {

  // print count every second
  while (1) {
    // Sleep for one second.
    chThdSleepMilliseconds(1000);
   if (debug) {
    // Print count for previous second.
    Serial.print(F("Count: "));
    Serial.print(count);

    // Print unused stack for threads.
    Serial.print(F(", Unused Stack: "));
    Serial.print(chUnusedStack(waThread0, sizeof(waThread0)));
    Serial.print(' ');
    Serial.print(chUnusedStack(waThread1, sizeof(waThread1)));
    Serial.print(' ');
    Serial.print(chUnusedStack(waThread2, sizeof(waThread2)));
    Serial.print(' ');
    Serial.print(chUnusedStack(waThread3, sizeof(waThread3)));
    Serial.print(' ');
    Serial.println(chUnusedHeapMain());
   }

   Serial1.println('uuuuuuuu');

    // Zero count.
```

```
  count = 0;
 }
}
//------------------------------------------------------------------------------
void setup() {
// pinMode(LED_PIN, OUTPUT);
//digitalWrite(LED_PIN, HIGH);

 if (debug) {
  Serial.begin(9600);
//   while (!Serial) {}
  Serial.println("Goodnight moon!");
 // read any input
 }

 delay(200);
//  while (Serial.read() >= 0) {}
 Serial1.begin(38400);
 while (!Serial1) {}
 if (debug) Serial.println("Goodmorning moon!");

 chBegin(mainThread);
 // chBegin never returns, main thread continues with mainThread()

 while(1) {}
}
//------------------------------------------------------------------------------
// main thread runs at NORMALPRIO
void mainThread() {

 // start sniffer thread
 chThdCreateStatic(waThread0, sizeof(waThread0),
                    NORMALPRIO + 4, Thread0, NULL);

 chThdCreateStatic(waThread1, sizeof(waThread1),
                    NORMALPRIO + 3, Thread1, NULL);

 // start blink thread
 chThdCreateStatic(waThread2, sizeof(waThread2),
                    NORMALPRIO + 2, Thread2, NULL);

 // start print thread
 chThdCreateStatic(waThread3, sizeof(waThread3),
                    NORMALPRIO + 1, Thread3, NULL);

 // increment counter
```

```
  while (1) {
    // must insure increment is atomic in case of context switch for print
    // should use mutex for longer critical sections
    noInterrupts();
    count++;
    interrupts();
  }
}
//------------------------------------------------------------------------------
void loop() {
 // not used
}
```

Final Notes

The production board 'V2' is loaded with the basic parts for a fully functioning KX3 BD/E. The schematic will show additional parts that are 'experimental'. They include additional driver pins, a pin for signaling mode to the Arduino and a serial switch for sampling data from the 'PC' side instead of the KX3 side. None of these capabilities have been programmed into the source code. Potential uses include: Command interpreter from the PC, commands to map the other output pins so that they can drive additional peripheral devices.

The basic desing is KISS (keep it simple stupid). If you are adventurous, you could set up your won Arduino development environment, load in the code and make changes. Please give me (NX1P) credit for the original! And please do not copy for a production model with out my permission.

20150531 copyright PDK, Paula Keezer NX1P