

```

package nmc.sidekick;

import java.io.*;
import javax.microedition.io.Connection;
import javax.microedition.io.Connector;
import javax.microedition.io.Datagram;
import javax.microedition.io.DatagramConnection;
import com ajile.drivers.spi.*;
import javax.comm.SerialPort;
import javax.comm.CommPortIdentifier;

import nmc.common.*;

public class CommandParser {

    //private static final String saIpAddr = "134.89.10.156";
    private static final String saIpAddr = "10.0.0.2";
    private static final int    saPortNumber = 8188;
    private static final int    skPortNumber = 8190;

    public IoResource[] channel;

    /**
     * Starts the application.
     * @param args an array of command-line arguments
     */
    public static void main(java.lang.String[] args) {
        CommandParser cp = new CommandParser();
        cp.loadAndGo(); // this is still the main thread;
        // we just transition to working on an instance
        // I am not sure whether the main thread should be
        // treated specially or not... at this point,
        // I am not treating it specially, and from here
        // it will go on to wait for a command, look it
        // up, and do it.
    }

    // This does not inherit from Runnable, so it is not a "real" run...
    public void loadAndGo() {

        System.out.println("Hello world.");

        channel = new IoResource[60]; // initial values are nulls

        initializePlatform();

        String hostIPAddress;
        String hostDatagramPort;
        int hostPortNumber;
        OutboundMessageQ outbound = null;
        InboundMessageQ inbound = null;
        // Set up the outbound and inbound queues
        try {
            outbound = new OutboundMessageQ(saIpAddr, saPortNumber);
            outbound.startOutboundMessaging();

            inbound = new InboundMessageQ(skPortNumber);
            inbound.startInboundMessaging();

        } catch (Exception e) {
            System.out.println("Caught exception while opening connection:"

```

```

                + e);
        }
        return;
    }

    for (int j=0; j<20; j++) {
        Message m = inbound.blockingGet();
        int msgType = m.getMessageType();
        switch (msgType) {
            case Message.SET_CURRENT_LIMIT_CMD: {
                SetCurrentLimitCmd curLimCmd = (SetCurrentLimitCmd)m;
                System.out.println("detected current limit cmd");
                IoResource ior = channel[curLimCmd.channelNumber];

                // The following is wordy, but it follows the IUnknown IDL
                // paradigm (sort of). You ask for the type of thing you
                // want. If you get back non-null, it is the right type
                // and you cast it to the correct type and you are there.
                // The thing might not have been the right type, but it could
                // get a hold of a thing of the right type. E.g., a dpa channel
                // can get a hold of an instrument.
                DpaBoard.DpaChannel aChannel =
                    (DpaBoard.DpaChannel)
                    ior.getResourceType(IoResourceType.DPA_BOARD);
                System.out.println("Got the dpa channel, doing set currentlimit");
                aChannel.setCurrentLimit(curLimCmd.currentLimit);
                AckReply r = new AckReply(curLimCmd);
                outbound.put(r);
            }
            break;
            case Message.POLL_CMD: {
                System.out.println("detected poll cmd");
                byte[] b = "Ding dong the witch is dead".getBytes();
                PollCmd p = (PollCmd)m;
                IoResource ior = channel[p.channelNumber];
                Instrument meow =
                    (Instrument) ior.getResourceType(IoResourceType.INSTRUMENT);
                byte[] dataOut = meow.poll();
                PolledDataReply pdr = new PolledDataReply(m,dataOut);
                outbound.put(pdr);
                System.out.println("Just did outgoing PolledDataReply: "
                    + pdr.toString());

                byte[] b2 = null;
                Message m2 = null;
                try {
                    b2 = pdr.cerealize();
                    m2 = DecerealizingFactory.parse(b2);
                } catch (IOException e) {
                    System.out.println("bad cereal! " + e);
                }
                System.out.println("Reconstruction: " + m2.toString());

            }
            break;
            case Message.INITIALIZE_INSTRUMENT_CMD: {
                System.out.println("detected INITIALIZE_INSTRUMENT_CMD");

                InitializeInstrumentCmd iic = (InitializeInstrumentCmd)m;
                System.out.println("    for channel number "
                    + iic.channelNumber);

                // We cannot use that yet, but in the real system
                // the DPA board index is channelNumber >> 1 and the

```

```

// low-order bit (channelNumber & 1) is the left/right
// discriminator

IoResource ior = channel[iic.channelNumber];

DpaBoard.DpaChannel aChannel =
    (DpaBoard.DpaChannel)
    ior.getResourceType(IoResourceType.DPA_BOARD);
System.out.println("Got the dpa channel, initializing it");
aChannel.initializeChannel();

InitializeInstrumentReply iir =
    new InitializeInstrumentReply(m, 90);
outbound.put(iir);
System.out.println("Just did outgoing "
    + "InitializeInstrumentReply"
    + iir.toString());
}
break;
default:
    System.out.println("unknown message received");
}
}
System.out.println("Around the loop " + j + " time(s).");
}

protected void initializePlatform() {
    /*
     * This is all the platform-specific configuration stuff.
     * Everything except the association between the instruments
     * and the port is likely to remain hardwired. The association
     * of instruments to serial lines must be configurable.
     */

    System.out.println("Starting 'initializePlatform()'");
    DpaBoard board = new DpaBoard(SpiMaster.SPI_SLAVE_SELECT_0);
    DpaBoard.DpaChannel aChannel = board.getLeftChannel();
    System.out.println("Finished creating left dpa channel");
    SerialPort cereal = null;
    try {
        cereal = openSerialPort("com1");
    } catch (IOException e) {
        System.out.println("could not open 'com1': " + e);
    }
    aChannel.setSerialPort(cereal);
    System.out.println("Finished creating the serial port on com1");
    Instrument guitar = new MicroCat();
    guitar.setSerialPort(cereal);
    aChannel.setInstrument(guitar);

    channel[0] = aChannel;

    System.out.println("Finished loading left channel");
    aChannel = board.getRightChannel();
    System.out.println("Finished creating right dpa channel");
    channel[1] = aChannel;
}

// this seems like it should go in some class of utility routines

```

```

protected SerialPort openSerialPort(String portName) throws IOException {
    CommPortIdentifier commPortId;
    SerialPort serialPort = null;
    try {
        commPortId = CommPortIdentifier.getPortIdentifier(portName);
        System.out.println("Whew, we got the port identifier for '"
            + portName + "'!");
        if (!commPortId.isCurrentlyOwned()) {
            System.out.println("Whew, no one owned " + portName);
            if (commPortId.getPortType() ==
                CommPortIdentifier.PORT_SERIAL) {
                System.out.println("Whew, com1 turns out to be a "
                    + "serial port...");
                try {
                    // the name seems to be completely meaningless.
                    // we use "xyz"
                    serialPort = (SerialPort)commPortId.open("xyz",
                        0);
                    // outStream = serialPort.getOutputStream();
                    // inStream = serialPort.getInputStream();
                } catch (javax.comm.PortInUseException e) {
                }
            }
        }
    } catch (javax.comm.NoSuchPortException e) {
        System.out.println("javax.comm.NoSuchPortException on 'com1'");
    }
    // This really should throw an exception if it cannot
    // create a fully functioning serial port.

    return serialPort;
}

```

02/24/02
16:03:26

nmc\sidekick\DataPort.java

1

```
package nmc.sidekick;  
  
public interface DataPort {  
    public void sendData(byte[] data);  
}
```

nmc\sidekick\DpaBoard.java

```

package nmc.sidekick;
import javax.comm.SerialPort;
import com.ajile.drivers.spi.*;
import java.lang.Thread;

// The DPA board looks like the following

/*
 * Major things not yet tested:
 * - taking a reading from the ADC
 * - taking all readings from the ADC
 * - turning each relay on, power cycle, first write is turn it off
 * - turning each relay off, power cycle, first write is turn it on
 * - turning each relay of, power cycle, first write is turn it off
 * - turning each relay on, power cycle, first write is turn it on
 * - I presume that interactions between relays are not significant
 * - getting an enabled interrupt
 * - getting an enabled interrupt and disabling it
 * - getting a disabled interrupt
 * - tripping the digital circuit breaker
 * - tripping the digital circuit breaker and resetting it
 * - repeatedly trip/reset the digital circuit breaker, deal with short
 * - various combinations of drive modes (unipolar/bipolar, RS-485, etc.)
 */

/*
- DPA
- Notes: In the listing below, single-bit register fields are only
  listed by their values for a '1' in that position. If a '0' in
  that position means something other than the simple negation of
  a '1', then the meaning of '0' is listed in the same line.
  There are only four items that are not single-bit fields: (1) the
  command (in all cases), (2) the power-down (or simply 'power') mode for
  an ADC Register Write, (3) the channel value for an ADC Register
  Write, and (4) the 'count', or number of clicks to move the
  potentiometer for a Dpot
  (Current Limit) Register Write. There are individual names for
  each possible value for the cmnd field and for the ADC power
  mode. The ADC Channel and the Dpot Count field are simply
  unsigned numbers, so only a mask value is specified for those.
- Defaults: Most of the values do not have meaningful defaults.
  Those that do have a '*' or a '!' before the value. An '*'
  means that the asterisked value is the default. A '!' means
  that a 0 in that bit position is the default.
- Summary of registers
  - ADC
  - Interrupt
  - Relays
  - Dpot
  - Channel Control
- DPA board-level commands (cmnd mask 0xf000, values:)
  - ADC
    - Notes: Programmer writes a command to the ADC and waits for
      the DPA's status to become not busy, which indicates
      that the command has been accepted and the conversion
      has been completed. After the DPA status becomes
      not busy, the programmer can read the ADC. If the power
      mode is power on, the command is also a conversion
      request; all bits are always meaningful, so every
      conversion request must include power mode, acquisition
      mode and drive types.
    - 0x4000 ADC Write (aka take reading)
      - Power Mode (mask 0x00c0, values:)
        - 0x0000: full power down [default when not sampling]
        - *0x0040: standby
        - 0x0080: power on / use internal clock [default when sampling]
        - 0x00c0: power on / use external clock
      - Acquisition Mode
        - !0x0020: External (0 means Internal)
      - SGL/DIF (drive type)
        - *0x0010: Single-ended (0 means Differential)
      - Uni/Bi (drive polarity)
        - *0x0008: Unipolar (0 means Bipolar)
      - Channels (mask 0x0007, values:)
        - 0x0007: Battery Supply Voltage (Vbat = 5.8*Vin)
        - 0x0006: Right Channel voltage sense (Vinstr=6*Vin)
        - 0x0005: Right Channel trip level sense
        - 0x0004: Right Channel current sense
        - 0x0003: Heat sink temperature
        - 0x0002: Left Channel voltage sense (Vinstr=6*Vin)
        - 0x0001: Left Channel trip level sense
        - 0x0000: Left Channel current sense
    - 0xb000 ADC Read (mask for values: 0x0fff)
- Interrupt Register
  - Notes: The DPA is capable of generating an interrupt to the
    SideKick processor ONLY on overcurrent conditions. If a
    channel has an overcurrent condition (is drawing more
    current than allowed by the SetCurrentLimit cmd), the
    electronic circuit breaker will trip. An interrupt will
    be asserted iff the electronic circuit breaker for that
    channel has tripped (and has not been reset), AND the
    interrupt enable for that channel is set, AND the global
    interrupt enable has been set. The assertion of an
    interrupt will cause three things to be true: the
    interrupt flag for that channel will be set, the global
    interrupt flag will be set, and the DPA board will
    assert an interrupt to the SideKick processor. The
    overcurrent condition itself is asserted by the digital
    circuit breaker, and that breaker is cleared by turning
    it off. (Reading the interrupt register has no effect
    on the interrupt flags.)
  - Commands
    - 0x5000 Write
    - 0xb000 Read
  - Register bits (mask 0x003f, RO=WriteOnly, RW=Read/Write)
    - 0x0020: (RW)Left Channel overcurrent intr en
    - 0x0010: (RW)Right Channel overcurrent intr en
    - 0x0008: (RW)Global intr en
    - 0x0004: (RO)Left Channel overcurrent intr flag
    - 0x0002: (RO)Right Channel overcurrent intr flag
    - 0x0001: (RO)Global intr flag
- DPA Channel-level commands (cmnd mask 0xf000, values:)
  - Relays
    - Commands
      - 0x2000: Write left channel register
      - 0x3000: Write right channel register
      - 0xa000: Read left channel register
      - 0xb000: Read right channel register
    - Register bits
      - 0x0004: Connect 485 Terminators

```

```

- 0x0002: Isolate communications ground
- 0x0001: Isolate instrument power
- Dpot (Digital Circuit Breaker Current Limit)
  - Commands
    - 0x0000: Write left channel
    - 0x1000: Write right channel
  - Register bits and fields
    - 0x0100: Save position
    - 0x0080: Up (move potentiometer UP)
    - 0x007f: mask for counter value (0 through 127)
- Channel Control
  - Commands
    - 0x6000: Write left channel
    - 0x7000: Write right channel
    - 0xd000: Read left channel
    - 0xe000: Read right channel
  - Register bits
    - 0x0020: Serial tx power on
    - 0x0010: Comm fast (0 means Slew Rate Controlled)
    - 0x0008: Comm Mode RS-485 (0 means RS-232)
    - 0x0004: Comm Half Duplex (0 means full duplex)
    - 0x0002: Comm Power On
    - 0x0001: Instrument Power On (this is the Digital Circuit Breaker)

*/

// Implementation notes: when a board is created (ctor) it should
// instantiate all the left and right components, because it should
// initialize them. From there on out, when a user of a channel
// manipulates any of its components, including components that are
// specialized to the left or right channel, they will do the right
// thing, since (a) they have access to the DPA board resources (shared
// between the channels) and (b) have their own control when that's
// appropriate. Since the operation will be specified in the (abstract)
// superclass while the implementation lives in the subclass, the
// user can safely call superclass operations and they will either
// do the right thing because they are identical for both channels, or
// they will do the right thing because the implementation that gets
// invoked has been correctly specialized to the correct channel.
// Probably the only thing that really needs to handle the left and
// right channels as such is the board itself, and it can do so.
// E.g., if it needs to initialize the current limit by setting both
// channels to zero, it can do { leftChannel.adc.setCurrentLimit(0);
// rightChannel.adc.setCurrentLimit(0); }.
//

public class DpaBoard {

    private DpaChannel dpaLeftChannel; // auto-initialized to null
    private DpaChannel dpaRightChannel; // auto-initialized to null
    private int[] adcReg = new int[1];
    private int[] leftRelayReg = new int[1];
    private int[] rightRelayReg = new int[1];
    private int[] leftChannelCtrlReg = new int[1];
    private int[] rightChannelCtrlReg = new int[1];
    private int[] interruptReg = new int[1];
    {
        adcReg[0] = 0;
        leftRelayReg[0] = 0;

```

```

        rightRelayReg[0] = 0;
        leftChannelCtrlReg[0] = 0;
        rightChannelCtrlReg[0] = 0;
        interruptReg[0] = 0;
    }

    /*
    public void printShadowRegisters() {
        System.out.println("adcReg[0] = " + adcReg[0]
            + "; leftRelayReg[0] = " + leftRelayReg[0]
            + "; rightRelayReg[0] = " + rightRelayReg[0]);
    }
    */

    abstract public class HdwrReg {
        protected int[] regVal;
        protected boolean changed = true; // power-up value uncertain

        HdwrReg() {
            System.out.println("HdwrReg(void) ctor invoked");
        }
        HdwrReg(int[] regVal) {
            System.out.println("HdwrReg(int[]) ctor invoked");
            this.regVal = regVal;
        }
        protected void set(int mask, int val) {
            if ((regVal[0] & mask) == val)
                return;
            regVal[0] &= ~mask;
            regVal[0] |= val;
            changed = true;
        }
        protected void set(int bitmask) {
            set(bitmask, bitmask);
        }
        protected void clear(int mask) {
            set(mask, 0);
        }
    };

    abstract public class Adc extends HdwrReg {

        public final static int Wr = 0x4000;
        public final static int Rd = 0xb000;

        private final static int PowerMask = 0x00c0;
        private final static int PowerOff = 0x0000;
        private final static int PowerStby = 0x0040;
        private final static int PowerOnIntClk = 0x0080;
        //private final static int PowerOnExtClk = 0x00c0;

        private final static int AcqExt = 0x0020;
        private final static int SglDrive = 0x0010;
        private final static int Unipolar = 0x0008;
        private final static int BatterySupplyVoltage = 0x0007;
        protected final static int RightVoltage = 0x0006;
        protected final static int RightTripLevel = 0x0005;
        protected final static int RightCurrent = 0x0004;
        private final static int HeatSinkTemp = 0x0003;
        protected final static int LeftVoltage = 0x0002;
        protected final static int LeftTripLevel = 0x0001;
        protected final static int LeftCurrent = 0x0000;

```

```

public Adc(int[] regVal) {
    super(regVal);
    System.out.println("Adc ctor invoked");
    // These values will ALWAYS be this way.
    setSglDrive();
    setUnipolar();
}
public void setPowerOff() {
    set(PowerMask, PowerOff);
}
public void setPowerStby() {
    set(PowerMask, PowerStby);
}
public void setPowerOnIntClk() {
    set(PowerMask, PowerOnIntClk);
}
// PowerOnExternalClock will NEVER be used. Here only for reference.
//public void setPowerOnExtClk() {
//    set(PowerMask, PowerOnExtClk);
//}
public void setAcqExternal() {
    set (AcqExt, AcqExt);
}
public void setAcqInternal() {
    set (AcqExt, 0);
}
public void setSglDrive() {
    set (SglDrive, SglDrive);
}
public void setDifferentialDrive() {
    set (SglDrive, 0);
}
public void setUnipolar() {
    set (Unipolar, Unipolar);
}
public void setBipolar() {
    set (Unipolar, 0);
}
// To take a reading, put the ADC into the on/ext state.
// Wait 100msec.
// Put it into the stby state.
// Wait for the board to become nonbusy.
// Read the value.
void initialize() {
    setAcqInternal();
    setPowerStby();
    writeReadSpi(Wr | regVal[0] | 0); // channel is don't-care here
    stallOnBusyBit();
}

protected int getReading(int adcChannel) {
    setAcqExternal();
    setPowerOnIntClk();
    writeReadSpi(Wr | regVal[0] | adcChannel);
    try {
        Thread.sleep(100);
    } catch (java.lang.InterruptedException e) {
        System.out.println("ADC sleep got interrupted: " + e);
    }
}

```

```

    stallOnBusyBit();

    setAcqInternal();
    setPowerStby();
    writeReadSpi(Wr | regVal[0] | adcChannel);
    stallOnBusyBit();

    int retval = writeReadSpi(Rd); // other 12 bits are don't-cares
    stallOnBusyBit(); // read cannot possibly make it busy, though
    return retval;
}

public int getBatterySupplyVoltage() {
    return getReading(BatterySupplyVoltage);
}
public int getHeatSinkTemp() {
    return getReading(HeatSinkTemp);
}
abstract public int getVoltage();
abstract public int getTripLevel();
abstract public int getCurrent();
};

public class AdcLeft extends Adc {
    public AdcLeft(int[] regVal) {
        super(regVal);
        System.out.println("AdcLeft ctor invoked");
    }
    public int getVoltage() {
        return getReading(LeftVoltage);
    }
    public int getTripLevel() {
        return getReading(LeftTripLevel);
    }
    public int getCurrent() {
        return getReading(LeftCurrent);
    }
}
};

public class AdcRight extends Adc {
    public AdcRight(int[] regVal) {
        super(regVal);
        System.out.println("AdcRight ctor invoked");
    }
    public int getVoltage() {
        return getReading(RightVoltage);
    }
    public int getTripLevel() {
        return getReading(RightTripLevel);
    }
    public int getCurrent() {
        return getReading(RightCurrent);
    }
}
};

public Adc adc; // Adc is abstract, a real DpaBoard must be
// created with a AdcLeft or an AdcRight, but
// the user does not need to know which

// To read the intr register, first do a read (otherwise you may get
// a stale value)
//

```

```
abstract public class RelayReg extends HdwrReg {
    private final static int Connect485Terminators = 0x0004;
    private final static int IsolateCommunicationsGround = 0x0002;
    private final static int IsolateInstrumentPower = 0x0001;
    RelayReg(int[] regVal) {
        super(regVal);
        System.out.println("RelayReg ctor invoked");
    }
    public void connect485Terminators() {
        set(Connect485Terminators);
    }
    public void disconnect485Terminators() {
        clear(Connect485Terminators);
    }
    public void isolateCommunicationsGround() {
        set(IsolateCommunicationsGround);
    }
    public void connectCommunicationsGround() {
        clear(IsolateCommunicationsGround);
    }
    public void isolateInstrumentPower() {
        set(IsolateInstrumentPower);
    }
    public void connectInstrumentPower() {
        clear(IsolateInstrumentPower);
    }
    abstract public void write();
};

public class LeftRelayReg extends RelayReg {
    LeftRelayReg(int[] regVal) {
        super(regVal);
        System.out.println("LeftRelayReg ctor invoked");
    }
    private final static int Wr = 0x2000;
    public void write() {
        writeReadSpi(Wr | regVal[0]);
        stallOnBusyBit();
    }
};

public class RightRelayReg extends RelayReg {
    RightRelayReg(int[] regVal) {
        super(regVal);
        System.out.println("RightRelayReg ctor invoked");
    }
    private final static int Wr = 0x3000;
    public void write() {
        writeReadSpi(Wr | regVal[0]);
        stallOnBusyBit();
    }
};

abstract public class ChannelCtrlReg extends HdwrReg {
    private final static int TxHiPowerOn = (1<<5);
    private final static int SlewRateNotLimited = (1<<4);
    private final static int CommMode485 = (1<<3);
    private final static int CommHalfDuplex = (1<<2);
    private final static int CommPowerOn = (1<<1);
    private final static int InstrumentPowerOn = (1<<0);

    ChannelCtrlReg(int[] regVal) {
```

```
        super(regVal);
    }
    public void setTxHiPower() {
        set(TxHiPowerOn);
    }
    public void setTxLoPower() {
        clear(TxHiPowerOn);
    }
    public void setSlewRateNotLimited() {
        set(SlewRateNotLimited);
    }
    public void setSlewRateLimited() {
        clear(SlewRateNotLimited);
    }
    public void setCommModeRs485() {
        set(CommMode485);
    }
    public void setCommModeRs232() {
        clear(CommMode485);
    }
    public void setCommHalfDuplex() {
        set(CommHalfDuplex);
    }
    public void setCommFullDuplex() {
        clear(CommHalfDuplex);
    }
    public void setCommPowerOn() {
        set(CommPowerOn);
    }
    public void setCommPowerOff() {
        clear(CommPowerOn);
    }
    public void setInstrumentPowerOn() {
        set(InstrumentPowerOn);
    }
    public void setInstrumentPowerOff() {
        clear(InstrumentPowerOn);
    }
    abstract public void write();
};

public class LeftChannelCtrlReg extends ChannelCtrlReg {
    LeftChannelCtrlReg(int[] regVal) {
        super(regVal);
        System.out.println("LeftChannelCtrlReg ctor invoked");
    }
    private final static int Wr = 0x6000;
    public void write() {
        System.out.println("doing a LeftChannelCtrlReg write()");
        writeReadSpi(Wr | regVal[0]);
        stallOnBusyBit();
    }
};

public class RightChannelCtrlReg extends ChannelCtrlReg {
    RightChannelCtrlReg(int[] regVal) {
        super(regVal);
        System.out.println("RightChannelCtrlReg ctor invoked");
    }
    private final static int Wr = 0x7000;
    public void write() {
        System.out.println("doing a RightChannelCtrlReg write()");
        writeReadSpi(Wr | regVal[0]);
    }
};
```

```

        stallOnBusyBit();
    }
};

abstract public class InterruptReg extends HdwrReg {
    public static final int Wr = 0x5000;
    public static final int EnableMask          = 0x0038;
    public static final int LeftOvercurrentEnable = 0x0020;
    public static final int RightOvercurrentEnable = 0x0010;
    public static final int GlobalOvercurrentEnable = 0x0008;
    public static final int FlagMask             = 0x0007;
    public static final int LeftOvercurrentIntrFlag = 0x0004;
    public static final int RightOvercurrentIntrFlag = 0x0002;
    public static final int GlobalOvercurrentIntrFlag = 0x0001;

    public InterruptReg(int[] regVal) {
        super(regVal);
    }

    public void setGlobalOvercurrentEn() {
        set(GlobalOvercurrentEnable);
    }
    public void clearGlobalOvercurrentEn() {
        clear(GlobalOvercurrentEnable);
    }
    public boolean isGlobalOvercurrentIntrFlagSet() {
        return ((regVal[0] & GlobalOvercurrentIntrFlag) != 0);
    }
    abstract public void setOvercurrentIntrEn();
    abstract public void clearOvercurrentIntrEn();
    abstract public boolean isOvercurrentFlagSet();
    public void write() {
        int retval = writeReadSpi(Wr | regVal[0]);
        regVal[0] = ((regVal[0] & EnableMask) | (retval & FlagMask));
        stallOnBusyBit();
    }
    //abstract public void read(); // actually, we can implement this here
};

public class LeftInterruptReg extends InterruptReg {
    public LeftInterruptReg(int[] regVal) {
        super(regVal);
    }
    public void setOvercurrentIntrEn() {
        set(LeftOvercurrentEnable);
    }
    public void clearOvercurrentIntrEn() {
        clear(LeftOvercurrentEnable);
    }
    public boolean isOvercurrentFlagSet() {
        return ((regVal[0] & LeftOvercurrentIntrFlag) != 0);
    }
};

public class RightInterruptReg extends InterruptReg {
    public RightInterruptReg(int[] regVal) {
        super(regVal);
    }
    public void setOvercurrentIntrEn() {
        set(RightOvercurrentEnable);
    }
    public void clearOvercurrentIntrEn() {
        clear(RightOvercurrentEnable);
    }
};

```

```

    public boolean isOvercurrentFlagSet() {
        return ((regVal[0] & RightOvercurrentIntrFlag) != 0);
    }
};

protected boolean doDebug = false;

protected int spiSlaveSelectValue;
protected SpiMaster spi = SpiMaster.getInstance();
protected boolean boardInitialized = false;

public DpaBoard (int spiSlaveSelectValue) {
    System.out.println("DpaBoard ctor invoked");
    this.spiSlaveSelectValue = spiSlaveSelectValue;
}

// NOTE: The create-if-null strategy will NOT work in the presence
// of contention by multiple threads
//
public DpaChannel getLeftChannel() {
    if (dpaLeftChannel == null)
        dpaLeftChannel = this.new DpaLeftChannel();

    return dpaLeftChannel;
}

public DpaChannel getRightChannel() {
    if (dpaRightChannel == null)
        dpaRightChannel = this.new DpaRightChannel();

    return dpaRightChannel;
}

public void toggleCs1() {
    spi.setSlaveSelect(spi.SPI_SLAVE_SELECT_1);
    spi.setSlaveSelect(spi.SPI_SLAVE_SELECT_NONE);
}
public void toggleCs2() {
    spi.setSlaveSelect(spi.SPI_SLAVE_SELECT_2);
    spi.setSlaveSelect(spi.SPI_SLAVE_SELECT_NONE);
}

protected int writeReadSpi(int dataOut) {
    spi.setSlaveSelect(spiSlaveSelectValue);
    int dataIn = spi.writeReadData(0xff & (dataOut >> 8));
    dataIn <= 8;
    dataIn |= (0xff & spi.writeReadData(0xff & dataOut));
    spi.setSlaveSelect(spi.SPI_SLAVE_SELECT_NONE);

    /**/
    System.out.println("    writeReadSpi(0x"
        + Integer.toHexString(dataOut)
        + ") returns 0x"
        + Integer.toHexString(dataIn));

    /**/

    // if (dataIn == 0xffff) or
    // (dataIn & 0x7fff) == 0x7fff) throw exception
    return dataIn;
}

```



```

public synchronized void stallOnBusyBit() {
    /*
    int regVal;
    do {
        regVal = writeReadSpi(0xf << 12);
    } while ((regVal & 0x7fff) == 0x7fff);
    toggleCs1();
    /**/
    /*
    System.out.println("Spi bus cycle complete");
    /**/
}

/*
 * The DpaChannel contains everything associated with the
 * channel, and also with its DPA board.
 */
public abstract class DpaChannel implements IoResource {

    protected boolean channelInitialized = false;
    protected boolean currentCtrlInitialized = false;
    protected Instrument instrument;
    protected javax.comm.SerialPort instrumentPort;
    protected int presentPosition;
    protected static final int MOVE_UP = 1;
    protected static final int MOVE_DOWN = 0;
    protected RelayReg relayReg;
    protected ChannelCtrlReg channelCtrlReg;
    protected InterruptReg interruptCtrlReg;

    public DpaChannel() {
        System.out.println("DpaChannel ctor invoked");
    }

    void initializeChannel() {
        if (! channelInitialized) {
            initializeBoard(); // no-op if board already initialized

            //      1) Initialize the ADC. Done in initializeBoard();

            //      2) Initialize the relays
            relayReg.disconnect485Terminators();
            relayReg.isolateCommunicationsGround();
            relayReg.isolateInstrumentPower();
            relayReg.write();

            //      3) Set Current Limit -- this is done by the
            //          SideArm (or some other configurator) later.

            setCurrentLimit(1400); // sets trip value for
            // digital circuit breaker
            // 1400 is 11 steps (rounded down)

            //      4) Configure the comm channel -- this is also done
            //          by the configurator.
            relayReg.connectInstrumentPower();
            toggleCs2();
            relayReg.write();
            System.out.println("initializing channelCtrlReg");
            channelCtrlReg.setTxHiPower();
            channelCtrlReg.setSlewRateNotLimited();

```

```

            channelCtrlReg.setCommModeRs232();
            channelCtrlReg.setCommFullDuplex();
            channelCtrlReg.setCommPowerOn();
            channelCtrlReg.setInstrumentPowerOn();
            channelCtrlReg.write();

            // should remember global setting and other channel's setting
            interruptCtrlReg.setOvercurrentIntrEn();
            interruptCtrlReg.write();

            channelInitialized = true;
        }
    }

    protected void initializeBoard() {
        System.out.println("first line of initializeBoard()");
        if (! boardInitialized) {
            System.out.println("Doing DpaChannel.initializeBoard()");
            spi.setClkDivider(spi.SPI_CLOCK_DIVIDER_64);

            spi.setSlaveSelect(spi.SPI_SLAVE_SELECT_NONE);
            spi.writeReadData(0); // Do a write with no chip select.
            // This forces the DPA into a known
            // state (just in case it had seen
            // spurious spi clock edges with a
            // spurious CS).

            adc.initialize();
            interruptCtrlReg.setGlobalOvercurrentEn();
            interruptCtrlReg.write();
            boardInitialized = true;
        }
    }

    public void setSerialPort(SerialPort aPort) {
        // if aPort is null or instrumentPort isn't, throw exception...
        this.instrumentPort = aPort;
    }

    public Object getResourceType(IoResourceType aType) {
        if (aType == IoResourceType.DPA_BOARD)
            return this;
        else if (aType == IoResourceType.INSTRUMENT)
            return instrument;
        else
            return null;
    }

    private void initializeCurrentCtrl() {
        System.out.println("in InitializeCurrentCtrl()");

        movePot(MOVE_DOWN, 99);
        stallOnBusyBit();
        presentPosition = 0;
        currentCtrlInitialized = true;
    }

    public void setCurrentLimit(int milliamps) {
        System.out.println("In setCurrentLimit(int milliamps)");
        if (! currentCtrlInitialized) {
            System.out.println(" about to initializeCurrentCtrl()...");

```

```

        initializeCurrentCtrl();
        System.out.println("  initializeCurrentCtrl() done.");
    }

    // if milliamps < 0 or > ?? throw exception

    // For safety, we always round DOWN (i.e., std integer division)
    // The device can be set in increments of 120 milliamps,
    // from 0*120 to 99*120 = 11880 (i.e., from 0 to 11.88 Amps)
    int newPosition = milliamps / 120;
    int diff = newPosition - presentPosition;
    if (diff > 0)
        movePot(MOVE_UP, diff );
    else if (diff < 0)
        movePot(MOVE_DOWN, -diff );

    stallOnBusyBit();

    this.presentPosition = newPosition;
    /*END*/
}

int getCurrentLimitCtrlValue() {
    return presentPosition * 120;
}

protected abstract void movePot(int direction, int nStepsUp );

public void setInstrument( Instrument anInstrument ) {
    instrument = anInstrument;
}

};

public class DpaLeftChannel extends DpaChannel {
    DpaLeftChannel() {
        System.out.println("DpaLeftChannel ctor invoked");
        adc = new AdcLeft(adcReg); // both channels share the adc reg
        relayReg = new LeftRelayReg(leftRelayReg); // but not relay reg
        channelCtrlReg = new LeftChannelCtrlReg(leftChannelCtrlReg);
        interruptCtrlReg = new LeftInterruptReg(interruptReg);
    }

    protected void movePot(int direction, int nStepsUp ) {
        System.out.println("DpaLeftChannel movePot(dir="
            + direction
            + "; steps=" + nStepsUp );

        int regValue =
            (0 << 12)           // CMND = 0 is left channel Dpot cmdnd
            | (direction << 8)  // up is 1 in bit 8, down is 0
            | (0 << 7)         // do not save position to flash
            | (nStepsUp & 0x7f); // the count is in the low order 7 bits

        writeReadSpi(regValue);
    }

};

public class DpaRightChannel extends DpaChannel {
    DpaRightChannel() {
        System.out.println("DpaRightChannel ctor invoked");
        adc = new AdcRight(adcReg); // both channels share the adc reg
        relayReg = new RightRelayReg(rightRelayReg); // but not relay reg

```

```

        channelCtrlReg = new RightChannelCtrlReg(rightChannelCtrlReg);
        interruptCtrlReg = new RightInterruptReg(interruptReg);
    }

    protected void movePot(int direction, int nStepsUp ) {
        int regValue =
            (1 << 12)           // CMND = 1 is right channel Dpot cmdnd
            | (direction << 8)  // up is 1 in bit 8, down is 0
            | (0 << 7)         // do not save position to flash
            | (nStepsUp & 0x7f); // the count is in the low order 7 bits

        writeReadSpi(regValue);
    }

};
}

```

02/24/02
16:03:26

nmc\sidekick\Instrument.java

1

```
package nmc.sidekick;

import javax.comm.SerialPort;

public interface Instrument {
    public void setSerialPort(SerialPort aPort);
    public void setDataPort(DataPort aDataPort);
    public byte[] poll();
    public void startStreaming();
    public void stopStreaming();
}
```

02/24/02
16:03:26

nmc\sidekick\IoResource.java

1

```
package nmc.sidekick;  
  
// this is just a test edit  
  
public interface IoResource {  
    public Object getResourceType(IoResourceType aType);  
}
```

02/24/02
16:03:26

nmc\sidekick\IoResourceType.java

1

```
package nmc.sidekick;

public class IoResourceType {
    private String name;
    private IoResourceType(String aString) {
        name = aString;
    }
    static final IoResourceType DPA_BOARD = new IoResourceType("DPA_CHANNEL");
    static final IoResourceType INSTRUMENT = new IoResourceType("INSTRUMENT");
}
```

```
package nmc.sidekick;

import java.io.*;
import javax.comm.SerialPort;

public class MicroCat implements Instrument, IoResource {

    SerialPort microcatLink;
    OutputStream ostream;
    InputStream istream;
    DataPort dataPort;

    public void setSerialPort(SerialPort aPort){
        this.microcatLink = aPort;
        try {
            ostream = aPort.getOutputStream();
            istream = aPort.getInputStream();
        } catch (IOException e) {
            System.out.println("error in getting io streams from serial port:"
                               + e);
        }
    }

    public void setDataPort(DataPort aDataPort){
        this.dataPort = aDataPort;
    }

    public byte[] poll(){

        byte[] cr = new byte[1];
        cr[0] = 0x0d;
        byte[] gt = ">".getBytes();
        System.out.println("gt is " + (char)gt[0]);
        int c = 0;
        int pos = 0;
        byte[] retbuf = new byte[1024];
        try {
            byte[] mchi = "\r".getBytes();
            ostream.write(mchi, 0, mchi.length);
            while (true) {
                int a = istream.available();
                c = istream.read();
                System.out.println("uC(" + a + "): " + (char)c);
                if ((char)c == '>')
                    break;
            }
            byte[] cmdstr = "ts\r".getBytes();
            ostream.write(cmdstr, 0, cmdstr.length);
            while (true) {
                c = istream.read();
                System.out.println("uC: " + (char)c);
                if (((char)c != '\r')
                    && ((char)c != '\n')
                    && ((char)c != '$')) {

                    if ((char)c == '>') {
                        if (pos > 7) {
                            break;
                        } else {
                            pos = 0;
                            continue;
                        }
                    }
                }
            }
        }
    }
}
```

```
        }
        retbuf[pos++] = (byte)c;
    }
} catch (IOException e) {
    System.out.println("error in talking to ucat: " + e);
}

byte[] realretval = new byte[pos];
for (int i=0; i<pos; i++) {
    realretval[i] = retbuf[i];
}

return realretval;
}

public Object getResourceType(IoResourceType aType) {
    if (aType == IoResourceType.DPA_BOARD)
        return null;
    else if (aType == IoResourceType.INSTRUMENT)
        return this;
    else
        return null; // probably should throw exception
}

public void startStreaming(){
    // launch a thread that gathers wool until it sees the
    // "send the wool to the woolmakers" tokens, and then goes
    // back to woolgathering...
    // It should probably be interrupted forcefully when
    // stopstreaming comes along.
}

public void stopStreaming(){
    // Interrupts the thread above, unless it is in the
    // phase of sending info... and even then, it should only abend
    // if it is partway through getting something sent, and that's
    // only because it might leave the Ethernet universe in an
    // uncertain state.
}
}
```

```
package nmc.sidekick;

import java.io.IOException;
import nmc.platform.FifoQueue;
import nmc.platform.DatagramIoPort;

import nmc.common.*;

/*
 * This requires its creator to have a live DatagramConnection
 * which is handed to the ctor.
 * The size of the queue is fixed now, but should be able to
 * grow.
 * We need a log that will keep track of major events like the
 * queue growing and major exceptions like the connection failing.
 * For a certain category of exceptional conditions we must fall
 * back and try to reestablish things, while not losing state
 * completely.
 */
public class OutboundMessageQ implements Runnable {
    private Object mutex; // because q can change (later)
    private FifoQueue q = new FifoQueue(50);
    private DatagramIoPort dgmTx;
    private Thread myThread;

    public OutboundMessageQ(String destinationIpAddr, int destinationPortNumber)
        throws IOException
    {
        dgmTx = DatagramIoPort.GetTxPort(destinationIpAddr,
            destinationPortNumber);

        /*
         fullDestAddr = "datagram://" + othersIPAddr + ":"
         + othersPortNumber;
        System.out.println("fullDestAddr is " + fullDestAddr);

        // Try to open the connection; it will be read and written and the
        // code here will handle timeouts.
        this.dgc = (DatagramConnection)Connector.open(
            fullDestAddr, Connector.READ_WRITE, true);
        */
    }

    public void put(Message msg) {
        // System.out.println("OutboundMessageQ.put ("
        // + msg.toString()
        // + ") called");
        q.enqueue(msg);
    }

    public void sendOne() throws IOException {
        System.out.println("Outbound doing blocking dequeue...");
        Message m = (Message)q.blockingDequeue();
        System.out.println("Outbound back from blocking dequeue...");
        System.out.println(" with message: " + m.toString());
        byte[] msgStr = m.cerealize();
        System.out.println(" msgStr.length is " + msgStr.length);
        dgmTx.send(msgStr);
        /*
        Datagram dg = dgc.newDatagram(124);
        dg.setData(msgStr, 0, msgStr.length);
        */
    }
}
```

```
        dgc.send(dg);
        */
        System.out.println("Just sent the outbound message");
    }

    public synchronized void startOutboundMessaging() {
        myThread = new Thread(this);
        myThread.start();
    }

    public void run() {
        System.out.println("OutboundMessage thread starting...");
        try {
            while (true) {
                sendOne();
            }
        } catch (IOException e) {
            System.out.println("Outbound caught exception, stopping. e="
                + e);
        }
    }

    /*
    public void sendThis(byte[] msgStr) throws IOException {
        Datagram dg = dgc.newDatagram(1024);
        dg.reset();
        dg.setData(msgStr, 0, msgStr.length);
        dg.setLength(msgStr.length);
        dgc.send(dg);
    }
    */
}
```