LabSmith uDevice Electrical Interface

Overview

LabSmith has developed a family of devices (uDevices) comprising active devices, including syringe pumps and valves, and sensors for modular microfluidics and miniature lab automation applications. These devices are designed to facilitate low-time-resolution device-function sequencing via external control, e.g., via a computer that is issuing commands to and polling for data from the devices. Such commands reach the uDevices via a two-line I2C interface (SCL and SDA). This I2C interface conforms to the physical-layer standards set by the Philips Corporation for 100 kb/s data transfer.

Electrical interface

Figure 1 shows a diagram of the connector pinout on the uDevice side and the 3M part number (151220-7422-TB) used on the uDevice. Power for uDevices must be supplied externally, including 5V (up to 1 A) and 12 V (up to 2A). A typical uDevice draws far less than the rated power. E.g., a syringe pump draws a maximum of ~200 mA from the 12 V supply and < 50 mA from the 5 V supply. The 5 V supply should not droop below 4 V or rise above 5.5 V. The 12 V supply should not droop below 10 V or rise above 13.6 V. The common current return line or ground is labeled "GND." For reduced analog sensing noise, a separate analog ground line (AGND) that draws little current is used. In the driving circuitry, this line should be connected to ground (0 V) in a way that current drawn on the GND lines do not couple offset voltages to the AGND line.

The electrical interface between uDevices is designed to support enhanced time resolution sequencing and controller-less sequencing via an array of eight shared digital signaling lines (D0—D7). These digital lines may be used for handshaking, signaling, or timing. In addition, a signal line "INT" has the ability to generate an interrupt for event-driven functions. A signal line "SYNC" can provide for high-accuracy time synchronization (better than ~1%) of all uDevices. If used by the uDevice firmware, this pin may be driven by a 32.768 kHz TTL oscillator. Current standard uDevice firmware versions do not use this synchronization line, however it is anticipated that future firmware versions will.

Four lines (D0, D1, D2, and D3) can be configured in firmware as analog sensing lines over the range 0 to 5 V. These lines are intended to be used in an application specific manner, such as to control the flow rate of a syringe pump uDevice via the analog output of a uDevice pressure sensor or external potentiometer, etc.
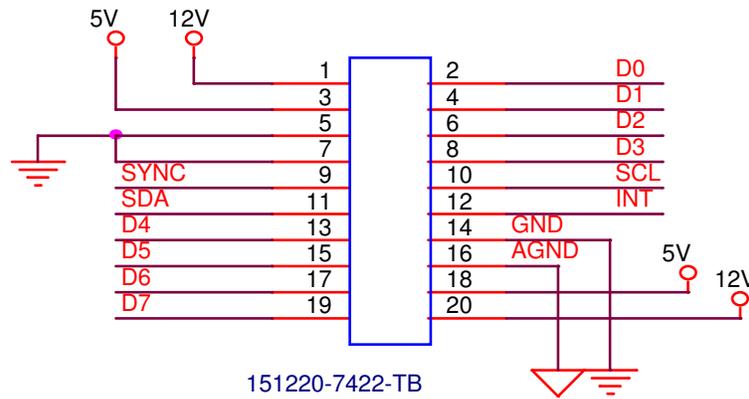
Figure 1 uDevice-side interface connector pinout.

The uDevices employ 18F-series PIC microcontrollers from Microchip. The uDevice firmware supports firmware updates over the I2C bus. The microcontrollers contain ~512 bytes of unused RAM, ~4k of unused instructions, and ~128 bytes of unused EEPROM space to support application-specific code.

I2C Packet Protocol

Each uDevice has a 7-bit address between 0x01 and 0x6F. These are saved in EEPROM and can be changed by commands issued over the I2C bus.

There are "Write" packets, in which a block of data or command is transmitted to a uDevice, and "Read" packets, in which a block of data is transmitted from a uDevice. A uDevice loads an internal buffer with the response to a "Write packet." The data in this buffer is transmitted from the uDevice during a "Read" packet. For this reason, a "Write packet" should always precede a "Read" packet. It is not necessary to issue a "Read" packet after every "Write" packet, however the response data from the previous "Write" will be overwritten and lost. It is good practice always to read the response to "Write" packets, since the response may indicate an error.

Multiple-byte integers are transmitted with the least-significant byte first.

Write Packets

To issue a packet to a device, begin with an idle I2C bus (SDA high, SCL high, all devices unaddressed). You may ensure this state exists by executing a "Stop" or P sequence according to the I2C spec. Begin the packet with an I2C "Start" sequence (S). The first transmitted byte is the I2C address, comprising the device address bit shifted to the left with a zero inserted in the LSB position. In accordance with the I2C spec, this initiates a write to a slave, in this case the uDevice. Next the master transmits the number of bytes (Cnt) that follow in the packet. Next the master transmits a one byte command (Cmd). If the command calls for data, the master transmits the data, finally the master transmits a Chksum and ends the packet with a stop sequence (P):

Write packet sequence:

```
S  / Addr<<1 | 0  <Ack>/ Cnt <Ack> / Cmd <Ack> /{optional data block
Data₁/<Ack>…DataN<Ack>}/Chksum<Ack>/ P
```

The <Ack>s are conditions where the uDevice holds the SDA line low during a 9th SCL pulse that follows the transmission of a byte. If a uDevice does not issue an `<Ack>` the packet should be aborted by issuing a `P` and retry-ed from the start. Under normal operation, the only time an `<Ack>` will not appear is when no device matches the address byte.

The uDevices employ clock stretching according to the I2C spec, in which they hold the SCL line low following an `<Ack>` until the received byte is processed. The master must wait for clock stretches following an `<Ack>` before attempting to transmit the next byte. The logic may look like this:

```
Release SCL (allow it to be pulled up by resistor)

While (SCL is low) wait

…
```

The checksum is calculated by subtracting all the previous bytes in the packet from zero and truncating the result to 8 bits, e.g., in the following function:

```c
void FormatCommand(unsigned char Addr, unsigned char*& pPkt, const unsigned
char* pData, unsigned int dataLength)
{
      unsigned char Chksum = 0;
      Chksum -= *(pPkt++) = Addr<<1;
      Chksum -= *(pPkt++) = (BYTE)(dataLength + 1);//Cnt
      unsigned* end = pPkt+dataLength;
      while (pPkt < end)
            Chksum -= *(pPkt++) = *(pData++);

      *(pPkt++) = Chksum;
}
```

Read Packets

Begin with the bus idle. Master perform a start sequence, followed by transmitting the slave (uDevice) address bit shifted to the left with a '1' inserted into the LSB. Addressed device shall `<Ack>` on 9th clock pulse.

Next the master transmits SCL pulses and shifts the SDA state in on the rising edge of the clock pulse according to the I2C spec. The uDevice controls the SDA line for the first 8 clock pulses. The master device holds SDA low during the 9th clock pulse, in other words, the master generates the acknowledgement `<MAck>`. The first byte is a byte-sized status token (`Tok`) having one of two valid values: `0xAA` means command executed; `0xEE` means command not executed. A command not executed usually indicates a checksum or other command packet error.

The master continues by a repeated process of transmitting SCL pulses and shifting bits in from the slave for 8 pulses and producing a `<MAck>` on the following pulse to read the balance of the packet. The next byte is the count (`RCnt`) of bytes that follow, including the checksum. If `RCnt` is zero, then the packet contains no checksum or data, only the status symbol (`0xAA` or `0xEE`) and the packet is complete. If `RCnt` is greater than zero, the master should repeat shifting `RCnt` bytes. The last byte is a checksum such that the sum of the received bytes following (not including) the status token is zero. The master terminates the packet by issuing a stop sequence (`P`).

Read packet sequence:

```
S  / Addr<<1 | 1  <Ack>/ Tok <MAck> / RCnt <MAck> /{data block
Data₁/<MAck>…Dataₙ<MAck>}/Chksum<MAck>/ P
```

If `RCnt > 0`, then `N`, the number of response data bytes is `RCnt - 1`.

Commands

As of the current firmware version, the commands are

Command	Cmd value	Target	Args	Buffered data (blank means ack or nak only)
CB_GETDATABLOCK	0x00	uDevice	None	Device dependent
CB_PING	0x01	uDevice	None	
CB_SETDEVADDR	0x02	uDevice	1-byte new address	
CB_GETVERSION	0x03	uDevice	None	2-byte firmware version 2-byte bootloader version 2-byte hardware version
RESERVED	0x04	SPS01		
CB_RESET	0x05	uDevice	None	
CB_STOP	0x06	uDevice	None	
CS_SETPERIOD	0x07	SPS01	3-byte period	
CV_SETVALVES	0x07	4VM01		
CS_MOVETOPOS	0x08	SPS01	2-byte position	
CS_GETMODE	0x09	SPS01	None	
CB_SETNAME	0x0A	uDevice	16 char uDevice "name"	
CB_GETNAME	0x0B	uDevice	None	16 char uDevice "name"

RESERVED	0x0C	SPS01		
CS_SETPOWER	0x0D	SPS01	1-byte power setting between 0x60 and 0xC0	
RESERVED	0x0E	uDevice		
RESERVED	0x0F	SPS01		
RESERVED	0x10	SPS01		
RESERVED	0x11	SPS01		
CB_SETCAL	0x12	uDevice	Device dependent	
CB_AUTOCAL	0x13	uDevice	None	
CB_GETCAL	0x14	uDevice	None	Device dependent
CS_SETDIAMETER	0x15	SPS01	2-byte unsigned integer	
CS_GETDIAMETER	0x16	SPS01	None	2-byte unsigned integer
RESERVED	0x17	SPS01		
CS_GETFACTORYCAL	0x18	SPS01		2-byte unsigned integer
CB_GETSERIALNUMBER	0x19	uDevice		2-byte "length" followed by "length" bytes
CB_GETSTATUS	0x1A	uDevice	None	1-byte motion status flags 2-byte position 2-byte micropulse count
CB_GETRAMBLOCK	0x1E	uDevice	1-byte RAM address 1-byte block count (16-bytes maximum)	Requested block of RAM
CB_SETRAMBLOCK	0x1F	uDevice	1-byte RAM address 1-byte block "cnt" (<=16 bytes) "cnt" bytes of data	
RESERVED	0xB5	uDevice		

uPS Pressure Sensor Pinout
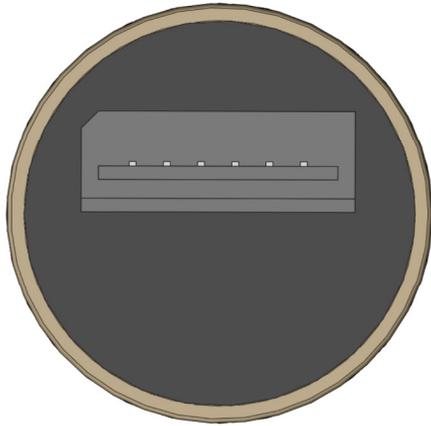
Pin	uPS Pressure Sensor
1	Digital Data (reserved)
2	Sensor PWR (3.3-5 V, low noise recommended)
3	Pressure out positive (positive leg of sensing bridge)
4	Pressure out negative (negative leg of sensing bridge)
5	Thermistor sense (ground referenced temperature-indicating, nonlinear signal)
6	Analog GND

The Pressure out negative and positive lie approximately at half the Sensor PWR voltage. The negative voltage decreases with increasing pressure. The positive voltage increases with increasing pressure. We recommend using differential amplification and A/D conversion. Temperature compensation is imperative where accuracy is important.

Additional Information

Could you give me more information on the serial commands (RS-232) to interface to the electrical interface (controller box)?

> The only difference in the packet that goes over the serial and the i2c is the serial package starts with a "%" sign and you don't need to poll the devices for a response. The EIB automatically read the device and forwards the response back over the serial port.
>
>
> I2C Example:
>
> S / Addr<<1 | 1 <Ack>/ Tok <MAck> / RCnt <MAck> /{data block Data1/<MAck>.DataN<MAck>}/Chksum<MAck>/ P
>
>
> RS232 Example (computer to device):
>
> '%' / Addr<<1 | 1 / Tok / RCnt /{data block Data1/.DataN}/Chksum
>
>
>
> RS232 Example Response:
>
> 0xAA / 0x00

I'm not getting the pic to relay my commands. What serial port configuration is correct? I'm using 57600, 8,n,1, no flow control. COM light on the EIB flashes, but I get no corresponding sda/scl toggles going out to the pump when I send a "%".

> Make sure the packet is sent as a single block (i.e., use one serial port block write to send a packet). There is a timeout in the packet receive routine in the EIB to help prevent hangs and permanent alignment errors. You have some inter-byte time, but you will not be able to step through byte writes will debugging, etc.
>
> The problem may be that you included the token '%' in your calculation of the checksum. You should be receiving a Nak 0xEE 0x00 back from the EIB over the com port because the checksum is bad.
>
> The correct checksum should be 256 – 2 (destination = device 1) – 26 (command is GETSTATUS) = 228.
>
> I hope this helps. Internally, here is the routine we use to format our outgoing packets. The calling function sets up the data payload at pdata and the length of the data payload in length and provides a buffer pd for the outgoing data (in this implementation, the function advances the buffer, but that's not important. What's important is the calculation of the checksum etc.

```
void CCommand::FormatCommand(BYTE dest, BYTE*& pd, int& i, const BYTE*
pData, DWORD length) {
      BYTE checksum = 0;
      BYTE* pStart = pd;

      *(pd++) = '%';  //first byte is the header token.  Note that the
header is not included in the checksum.
                                //this is only used as a
packet-alignment indication for the EIB and is stripped from the
                                //packet before the EIB forwards it over
the I2C bus.
      checksum -= *(pd++) = dest<<1; //next byte is the destination
address * 2
      checksum -= *(pd++) = (BYTE)(length + 1); //next byte is the
length of bytes to follow (+1 for checksum)
      BYTE* end = pd+length;
      while (pd < end)
//memcpy with checksum update
            checksum -= *(pd++) = *(pData++);

      *(pd++) = checksum;
//now add the checksum.
      i = pd - pStart;  //count of bytes in formatted packet
      //Now you should write the packet of i bytes starting at pStart
to the serial port.
}
```

How would I specify/query the flow rate that my movement would occur at?

The conversions from volume and flow-rate to counts are usually done in the SDK. I included a few critical functions below. Note that the implementation details are subject to change. Note that the CCommand(...) constructor calls the FormatCommand function I sent you previously. The second arg is the data payload and the third arg is the length of the data payload.

```
#define USYR_ENCODER_SIZE 4096//this is changing in a future firmware
release to 65536 const double CSyringe::m_dPositionScale = 13.0; //mm
of fictitious full stroke.
            //can get alternatively read this from SPS01
#define CS_MAX_PERIOD         655350        //Clock cycles between micro
steps
#define CS_MIN_PERIOD         108
#define CS_CLOCK_FREQ         41943040/32*44.59 //counts/second

bool CSyringe::CmdMoveToVolume(double dVolume)//dVolume is in ul {
      if (m_dDiameter == 0.0) return false;

      const double coeff = 1.27324062 * USYR_ENCODER_SIZE;
      int nPos = (int)(coeff * dVolume
/(m_dDiameter*m_dDiameter*m_dPositionScale) + 0.5);
      return CmdMoveToPosition ( nPos + m_nOutStop ); }

double CSyringe::GetVolumeFromPos(int pos)const {
```

```cpp
        const double coeff = 0.7853975/USYR_ENCODER_SIZE;
        return coeff * m_dDiameter*m_dDiameter * m_dPositionScale * (pos-
m_nOutStop); }

#define CB_GETCAL           0x14
bool CSyringe::CmdGetCal()//need to call this to get m_nOutStop.
Usually only once during // initialization {
        BYTE  command[12] ;
        BYTE  *pc = &command[0] ;

        if(!m_pInterface) return false;

        *(pc++) = BYTE( CB_GETCAL ) ;

        CCommand* pCommand = new CCommand( m_nAddress, command, 1,
"Reading Syringe End Points..." ) ;
        bool retval = m_pInterface->DoCommand( pCommand ) ;

        if( retval )
        {
                short int nOutStop,nInStop;
                BYTE* pd = &pCommand->m_pResponse[2] ;
                nOutStop = *(pd++) ;
                nOutStop |= *(pd++)<<8 ;
                nInStop = *(pd++) ;
                nInStop |= *(pd++)<<8 ;

                m_nInStop = nInStop; //copy over atomically to prevent
multithreading issues
                m_nOutStop = nOutStop;

        }
        else
        {
                //don't change the previous cal values
        }

        delete( pCommand ) ;

        return retval;
}

#define CS_SETPERIOD 0x07
bool CSyringe::CmdSetSpeed(double dSpeed) //dSpeed is flow rate in
ul/min {
        if (m_dDiameter <= 0.0) return false;
        if (!m_pInterface) return false;


        const double cSpeed = (CS_ACTUATION_DIST * CS_CLOCK_FREQ *
0.04908738521234);
        int nPeriod = (int)(cSpeed * m_dDiameter * m_dDiameter / dSpeed
+ 0.5);

        int maxper = 0xFFFFFF;
```

```
            if (nPeriod < CS_MIN_PERIOD ) nPeriod = CS_MIN_PERIOD;
            else if ( nPeriod > maxper) nPeriod = maxper ;

            BYTE command[12];
            BYTE* pc = &command[0];
            *(pc++) = BYTE(CS_SETPERIOD);
            *(pc++) = nPeriod&0xFF;
            *(pc++) = (nPeriod>>8)&0xFF;
            *(pc++) = (nPeriod>>16)&0xFF;

            CCommand* pCommand = new CCommand(m_nAddress,command, 4, "Setting
    speed...");
            bool retval = m_pInterface->DoCommand(pCommand);
            delete pCommand;

            return retval;
    }
```

What is CS_ACTUATION_DIST?

```
    #define CS_ACTUATION_DIST     0.02 //mm/step
```

What are the standard syringe sizes for the SPS01?

```
            m_dStdDia[0] = 0.729;
            m_dStdVol[0] = 4.0;
            m_csName[0]= _T("4 ul");

            m_dStdDia[1] = 1.031;
            m_dStdVol[1] = 8.0;
            m_csName[1] = _T("8 ul");

            m_dStdDia[2] = 1.458;
            m_dStdVol[2] = 20.0;
            m_csName[2] = _T("20 ul");

            m_dStdDia[3] = 2.304;
            m_dStdVol[3] = 40.0;
            m_csName[3] = _T("40 ul");

            m_dStdDia[4] = 3.256;
            m_dStdVol[4] = 80.0;
            m_csName[4] = _T("80 ul");
```

How do I read/interpret sensor measurements from the 4AM?

You retrieve the data from the 4AM via sending the CM_GETSTATUS command. The
4AM returns one status byte followed by four sets of 3-byte integers that
contain the readings, followed by four bytes that describe the regulation
status of each sensor. The 4AM can sample a variety of different sensors,

each having its own scale factor between 4AM-reported readings and standard units. Here is sample code for polling the status and interpreting the answer:

```cpp
bool C4AM::CmdGetStatus()
{
      if (!IsOnline()) return false;
      //this is an efficient state poll.
      BYTE  command[12];
      BYTE  *pc = &command[0];

      *pc = BYTE(CB_GETSTATUS);

      CCommand* pCommand = m_pInterface->LockCommand();
      pCommand->Create(m_nAddress, command, 1, _T("Reading 4AM Status..."));
      pCommand->SetTimeout(500);
      bool retval = m_pInterface->DoCommand();

      if (retval)
      {
            unsigned char* pdata = &pCommand->m_pResponse[2];
            m_Status = *(pdata++);
            m_bIdle = ((m_Status&0x80) == 0);
            if (m_pS1)
                  m_pS1->SetRawStatus(pdata);
            pdata += 3;
            if (m_pS2)
                  m_pS2->SetRawStatus(pdata);
            pdata += 3;
            if (m_pS3)
                  m_pS3->SetRawStatus(pdata);
            pdata += 3;
            if (m_pS4)
                  m_pS4->SetRawStatus(pdata);
            pdata += 3;


            bool bDone = true;
            //next come 4 bytes of reg status
            BYTE bStatus = *(pdata++);
            if (m_pS1)
            {
                  m_pS1->SetRegStatus(bStatus);
            }

            bStatus = *(pdata++);
            if (m_pS2)
            {
                  m_pS2->SetRegStatus(bStatus);
            }

            bStatus = *(pdata++);
            if (m_pS3)
            {
                  m_pS3->SetRegStatus(bStatus);
            }
```

```
                bStatus = *(pdata++);
                if (m_pS4)
                {
                        m_pS4->SetRegStatus(bStatus);
                }
        }

        m_pInterface->UnlockCommand();

        return retval;
}

void CSensor::SetRawStatus(unsigned char* pdata)
{
        int i = 0;
        i |= *(pdata++);
        i |= *(pdata++)<<8;
        char t = *(pdata++);
        i |= t<<16;
        if (t &0x80)//convert from 24-bit signed int to 32-bit signed int
                i |= 0xFF000000;
        double dStdValue = ConvertDeviceUnitsToStdReading(i);
        double dStdValue = p*m_dGainScale;
        //dStdValue is in kPa.  From here convert to desired units

}

double CSensor::ConvertDeviceUnitsToStdReading(int r)const
{

        const double igscale = 1.0/(256.0*256.0*128.0);

        double rat = double(r);
        rat *= igscale;
        rat *= GetPressureRange_kPa();//returns full-scale pressure in kPa,
        //    e.g., 250kPa sensor returns 250.0;
        return rat;
}
```