Constant Performance PCB Pump Driver:

Communications & Commands

IDEX HEALTH & SCIENCE

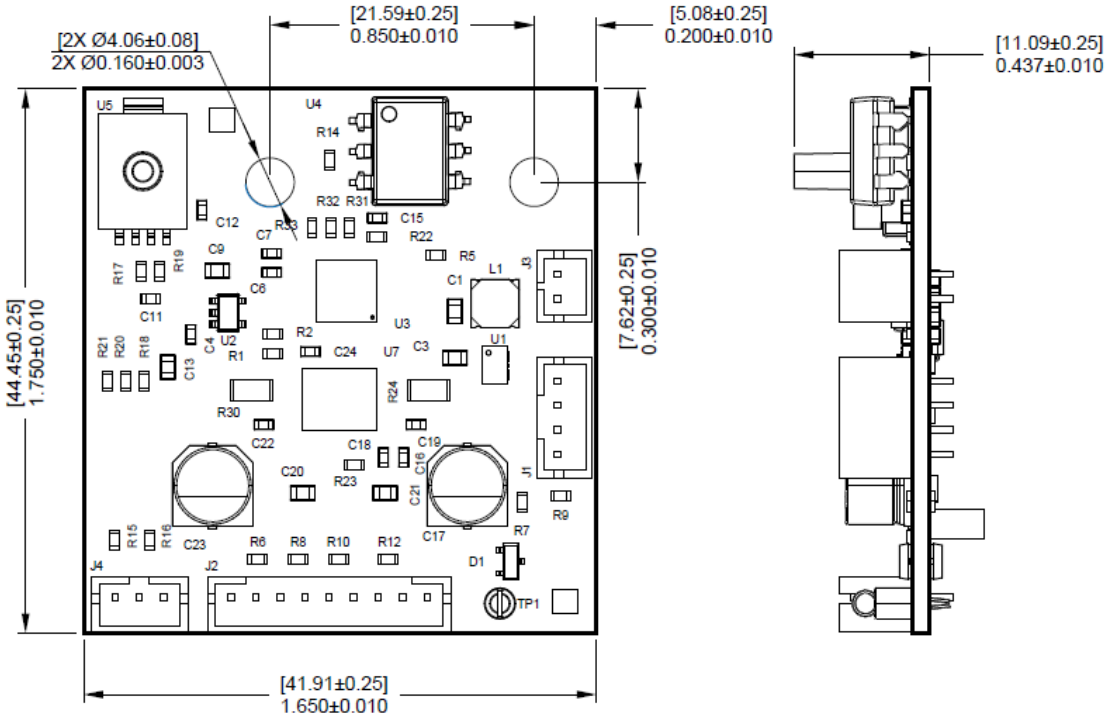
Intelligent Solutions for Life®

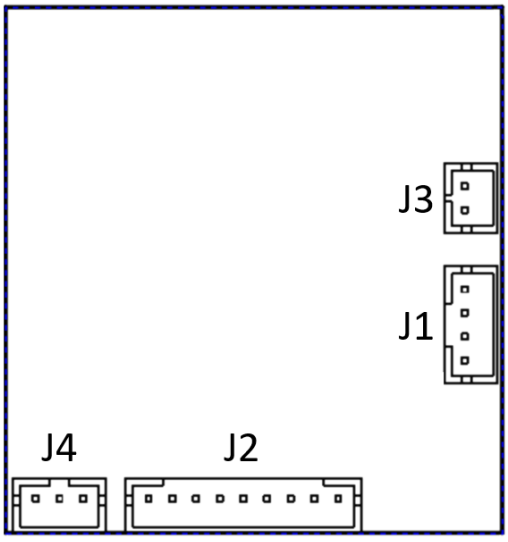
Fluidics | Optics | Consumables | Assemblies

Table of Contents

Board Dimensions and Connector Locations

Dimensions are in [millimeters] and inches. For Reference ONLY



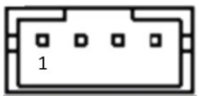


PCB Connections

Connector	Function	PCB Part Number	Mating Part Number
J1	Power Input	JST B4B-PH-K-S(LF)(SN)	PHR-4
J2	Communication (I²C and UART)	JST B9B-PH-K-S(LF)(SN)	PHR-9
J3	Error Output	JST B2B-PH-K-S(LF)(SN)	PHR-2
J4	LED Indicators	JST B3B-PH-K-S(LF)(SN)	PHR-3

J1 – Power Input

P1	P2	P3	P4
+24VDC*	+24VDC*	GND	GND



*Input voltage on J1 can be from +15VDC to +24VDC

J2 – Communication (I²C and UART)

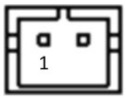
P1	P2	P3	P4	P5	P6	P7	P8	P9
I2C_SDA	I2C_SCL	U1_TX	U1_RX	U0_RX	U0_TX	GND	ISPn	RESETn

I2C_SDA, I2C_SCL: Open drain I2C pins
U1_RX, U1_TX: 3.3V UART pins, 5V tolerant
U0_RX, U0_TX, ISPn: FACTORY USE ONLY – DO NOT CONNECT.
RESETn: 3.3V logic pin, 5V tolerant. Low signal resets the board

J3 – Error Output

P1	P2
Emitter	Collector

Emitter, Collector: H11G2SR2M Opto-coupler output pins



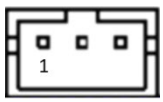
Optically isolated output:
Off (non-conducting) when vacuum is ≤ 30mmHg above setpoint, and no error is detected
On (conducting) when vacuum is > 30mmHg above setpoint, or an error is detected

Stepper motor driver is turned off when an error occurs and requires either power cycling or a low pulse on the RESET input to restore normal operation.

J4 – LED Indicators

P1	P2	P3
Red LED anode	GND	Green LED anode

Flashing Amber: Vacuum > 30mmHg above setpoint.
Flashing Green: Vacuum ≤ 30mmHg above setpoint OR Vacuum < setpoint.
Solid Green: Vacuum at setpoint.
Solid Red: Error (any other state than those listed above)
Alternating Green/Red: "OFF" command issued to PCB



Hardware Interfaces

Two hardware interfaces are supported, I²C and UART (3.3V, 5V tolerant).

Interface	Point to Point or Multidrop	Encoding	Speed (max bits/sec)
I²C	Multidrop	Binary	100K
UART	Point to Point	ASCII	115.2K

I²C

I²C is a multidrop standard that supports multiple units on the same bus. The individual units are separately addressable. The implementation follows the I²C standard for Start, Write Address, Read Address, Acknowledge, No Acknowledge, and Stop conditions.
Addresses in I²C are encoded into the upper 7 bits of the first byte. The last bit of the first byte is the Read/Write bit, set to 0 for a Write or 1 for Read. For example, if the unit address is 9, then the address byte for I²C will be 18 (0001 0010) for write, and 19 (0001 0011) for read.
The packet encoding for I²C is binary.

UART

UART is a point to point standard where only one unit can be connected to the UART port on the master.

UART is similar to RS-232 except for the signaling levels.

Caution – **DO NOT CONNECT TO A STANDARD RS-232 PORT or DAMAGE WILL OCCUR!**

The levels for UART are standard 3.3V CMOS logic levels. Input thresholds are:
V_{IL} = 0.99V (3.3V * .3)
V_{IH} = 2.31V (3.3V * .7)

The input is 5V tolerant so you can drive it from 0V to 5V with no damage.
Output levels are:
V_{OL} = 0.4V
V_{OH} = 2.9V

Addresses are included when using the IDEX command set. The address must match the unit's address, or the command will be ignored.

The packet encoding for UART is ASCII.

Command Packet Format

Commands packets have this format:

Send to device

Address, Length, Command Code, Device address, [Arguments], CRC

- Address = 4 to 123, 0 is reserved for broadcast messages to all units, (default address is 9)
- Length = byte count of the message including the length byte and the CRC bytes (does not count the address byte)
- Command Code = 1 byte command code (from tables below)
- Device address = sub address within this unit (always 0)
- Arguments = optional arguments as required by the command
- CRC = 2 byte CRC-CCITT

Response from device

Command Status, Length, [Data], CRC

- Command Status = 0 is successful status, >0 is a failure (see Command Status table below)
- Length = byte count of the message including the length byte and the CRC bytes
- Data = returned values, if any
- CRC = 2 byte CRC-CCITT

Command Status:
0 – command completed
4 – bad CRC
5 – bad command
8 – parameter unknown
12 – missing start character
13 – incorrect packet size
14 – command timeout
15 – no carriage return
16 – non-hex character

Packet Encoding

Binary Encoding

This type of encoding, used by the I²C interface, is simple to understand. You send the packet (address, length, command, parameters, CRC) as binary bytes to the I²C port. The PCB is an I²C slave so it cannot send a response unless the master asks to read it. It is not necessary to ask to read the slave if you don't care about the status. If you do want to read the status you must issue the read address and then clock in the response following the standard I²C protocol.

Here is an example of the IDEX command to turn off the pump and the response from the unit (over I²C at address 9): (gray are bytes sent from master, yellow are bytes sent from unit)

0x12	0x06	0x55	0x00	0x00	0x2B	0xD7
Address write	Number of bytes	Command code	Always 0	Value (off)	CRC1	CRC2

0x13	0x00	0x03	0x2D	0x6C
Address read	Command Status	Number of bytes	CRC1	CRC2

Here is an example of the IDEX command to set the flow rate to 5 ml/min and the response from the unit:

0x12	0x09	0x7E	0x00	0x00	0x4C	0x4B	0x40	0x77	0xFA
Addr write	Length	Command code	Always 0	Value (5,000,000 nL/min)				CRC1	CRC2

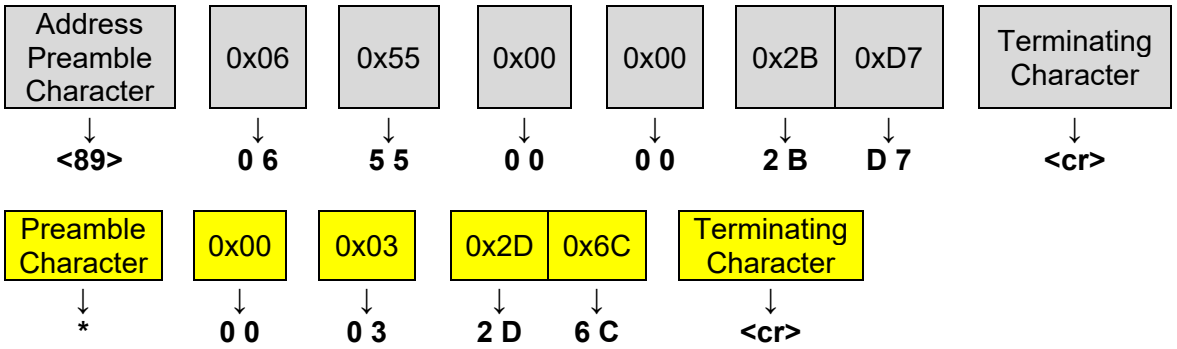
0x13	0x00	0x03	0x2D	0x6C
Address read	Command Status	Number of bytes	CRC1	CRC2

ASCII Encoding

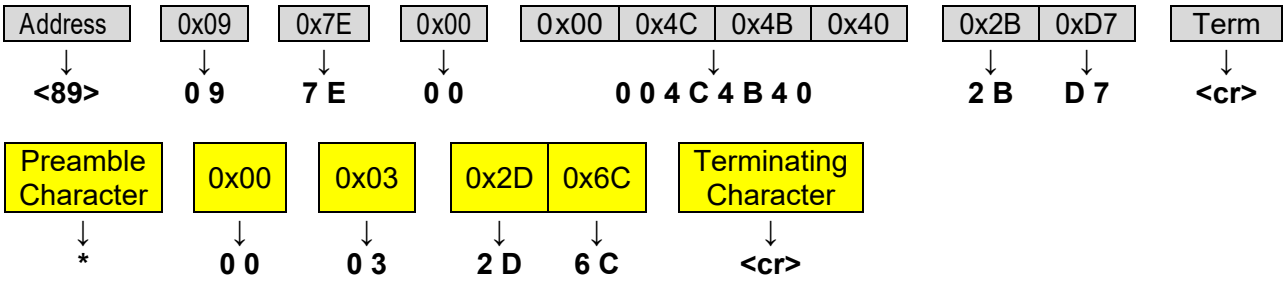
ASCII Encoding, used by the UART interface, sends all data after the address encoded as a string of ASCII characters. It replaces the address byte with a preamble byte that encodes the address as address value + 128 (shown below as address 9 + 0x80). It then encodes the rest of the message as ASCII hex digits ('0' to '9', 'A' to 'F'). Finally, a carriage return terminates the packet. The string of characters transmitted is shown at the bottom of the diagram below.

Since a UART can receive and transmit independently, it will always send a response to any command it receives. The board receives packets on the U1_RX pin (J2-pin4) and send the response on U1_TX (J2-pin3). The response always begins with the "*" character (0x2A) and ends with a carriage return.

Here is the same IDEX Pump Off command and response sent over UART with ASCII encoding:



Here is an example of the IDEX command to set the flow rate to 5 ml/min and the response from the unit:



IDEX Command Set

System Commands

Command	Code	Arguments	Return Values
Get vendor name	0x21	-	'IDEX'
Get firmware part number	0x22	-	Up to 9 ASCII characters with a terminating null
Get firmware revision	0x23	-	2 ASCII chars: Major revision, Minor revision
Get System part number	0x24	-	Up to 9 ASCII characters with a terminating null
Get System serial number	0x26	-	Up to 10 ASCII characters with a terminating null
Get System revision	0x29	-	2 ASCII chars: Major revision, Minor revision
Get mfg date	0x2B	-	Mfg Year – 2000 Mfg Month 1 – 12 Mfg Date 1-31
Set Board Address	0x2D	1-byte, valid range 4 to 123	-
Reset	0x2E	-	-
Get Command Status	0x30	-	Status byte
Set baud rate	0x33	1 - 9600, 2 – 19200, 3 38400, 4 – 57600, 5 -115200	Returns a standard acknowledgement packet at the current baud rate and then switches to the new baudrate for the next command
Get baud rate	0x35	-	Returns a code representing the Baud rate: 1 - 9600, 2 – 19200, 3 38400, 4 – 57600, 5 -115200
Load Default Parameters	0x38	-	Restores the factory default parameters
Save Parameters	0x39	-	Writes the operating parameters to non-volatile memory
Get PCBA Part #	0x3A	-	9 ASCII characters with a terminating null
Get Parameter	0x3F	1-byte parameter number	4-byte parameter value
Set Parameter	0x40	1-byte parameter number, 4 byte parameter value	-
Pump On/Off control	0x55	0 (off) or 1 (on)	-

Command	Code	Arguments	Return Values
Get vacuum	0x72	-	Returns the vacuum pressure in millimeters of mercury * 10
Get Status	0x79	Argument 1: Number of int16 values to read. Argument 2: Starting index into table. Example: 2, 1 would return the Vacuum and Average motor speed.	Returns an array of int16 the specified length. 0: Sys state (0-off, 1-low pressure, 2-at setpoint, 3-hi pressure, 4-very hi pressure, 5-fault) 1: Vacuum xxx.x mmHg 2: Average motor speed xxx.x rpm 3: Pulsation xxx.x 4: Pressure delta xxx.x mmHg 5: Instantaneous motor speed xxx.x rpm 6: PID error xxx.xx mmHg 7: Instantaneous Vacuum xxx.xx 8: ADC reading xxxxx counts 9: PID proportional xxx.x 10: PID integral xxx.x
Get PCBA Serial #	0x7A	-	Up to 10 ASCII characters with a terminating null
Get PCBA Revision	0x7C	-	2 ASCII chars: Major revision, Minor revision
Set Flow Rate	0x7E	4-byte value in nL/min, range is 1 to 10,000,000 nL/min	-
Set Standby	0x80	0 – return to normal operation at previous vacuum level 1 – set vacuum level to 288 mmHg	-
Set System part number	0x25	Up to 9 ASCII characters with a terminating null	-
Set System serial number	0x28	Up to 10 ASCII characters with a terminating null	-
Set System revision	0x2A	2ASCII characters: Major revision, Minor revision	-

Device Parameter Numbers

The parameters are read or written by the Get Parameter (0x3F) and Set Parameter (0x40) commands. All parameter arguments are read or written as 32-bit values (4 bytes) even if the actual parameter range is smaller.

Parameter Num	Parameter	Units
88 (0x58)	Vacuum set point	1/10 mmHg
89 (0x59)	Ambient atmospheric pressure	1/10 mmHg
90 (0x5A)	Efficiency 60% to 90%	%
94 (0x5E)	Pump down timeout	Seconds
95 (0x5F)	Error timeout	Seconds

Programming Examples

The following code examples will help in understanding how to develop a communication program.

CRC-CCITT Example

The CRC is calculated starting with the address and continuing through the entire message. Then the CRC bytes are appended to the end of the message to form the complete packet. This code will be called by other functions later in the examples.

```
uint16_t AccumulateCRC(uint8_t data, uint16_t crc)
{
  crc = (uint8_t)((inputCrc >> 8) | (inputCrc << 8));
  crc ^= data);
  crc ^= (uint8_t)(crc & 0xFF) >> 4;
  crc ^= (crc << 8) << 4;
  crc ^= ((crc & 0xFF) << 4) << 1;
  return crc;
}

void AppendCrc(uint8_t[] buffer)
{
  uint16_t crcWord = 0xFFFF;
  int i;
  crcWord = AccumulateCRC(addr, crcWord);
  crcWord = AccumulateCRC(num, crcWord);
  crcWord = AccumulateCRC(cmd[0], crcWord);
  crcWord = AccumulateCRC(dev, crcWord);
  for (i = 0; i < buffer[1]-2; i++)  // loop thru buffer minus the CRC
  {
      crcWord = AccumulateCRC(buffer[i], crcWord);
  }
  buffer[i] = (crcWord >> 8) & 0xFF;   // CRC upper byte
  buffer[++i] = crcWord & 0xFF;              // CRC lower byte
}
```

Create A Message

The following code will create a message to be sent to a board at the default address 9. The message is created in the tx_buffer, then passed to the CRC code to calculate and append the CRC bytes.

This simple example directly creates the command packet. In a real program you would create a more general solution by passing in the cmd and any required arguments and put them together in the buffer.

```
uint8_t addr = 9;           // default address
uint8_t tx_buffer[18];// large enough to hold longest packet
uint8_t output_buf[38]; // large enough to hold the longest ASCII encoded packet

 void ComposeMessage()
 {
   uint8_t cmd = 0x7E;       // cmd to set flow rate
   uint8_t dev = 0;          // device is always 0
   uint32_t flow = 5000000;  // flow rate 5 mL/min (units is in nL/min)
   uint8_t length = 5 + 4;   // minimum length is 5 (len + cmd + dev
                             // + 2 CRC), + 4 byte argument
   int i;
   // compose the basic message, break the flow argument into 4 bytes
   uint8_t msg[8] = {addr,length,cmd,dev,(flow>>24)&0xFF, (flow>>16)&0xFF,
                (flow>>8)&0xFF, flow&0xFF};
   for (i=0; i<8; i++)       // transfer the msg bytes into the tx_buffer
     tx_buffer[i] = msg[i];
   AppendCrc(tx_buffer);
 }
```

Here is an example to create an I²C message. After calling the above code to compose the basic binary message, all it needs to do is adjust the address to form a real I²C Write address, then copy the whole packet into an output buffer, ready to send.

```
void I2CMessage()
{
    int i;
    ComposeMessage();                 // create the basic message with the CRC added
    output_buf[0] = addr << 1; // shift addr left 1 to create I2C write address
    for (i = 1; i < tx_buffer[1]; i++)
    {
        output_buf[i] = tx_buffer[i]; // copy rest of message into output
    }
    // Here you would call code to transmit the output_buf out the I2C port
}
```

Create a UART Packet

UART is only slightly more complicated because it needs to ASCII encode all the packet after the address, and add a carriage return to the end. Note in the following code how it starts by using the same ComposeMessage code and then also calls the code to convert binary bytes into hexadecimal encoded bytes.

```
uint8_t ConvertToHexadecimal(uint8_t c)
{
  if (c < 10)
      c += 0x30;  // convert to ASCII characters "0" thru "9"
  else
      c += 0x37;  // convert to ASCII characters "A" thru "F"
  return c;
}

uint16_t bin2hex(uint8_t data)
{
  uint16_t hex;
  uint8_t c;
  hex = ConvertToHexadecimal(data >> 4) << 8; // convert upper 4 bits
  hex |= ConvertToHexadecimal(data & 0xF); // convert lower 4 bits
  return hex;
}

void ComposeMessage()
{
  uint8_t cmd = 0x7E;       // cmd to set flow rate
  uint8_t dev = 0;          // device is always 0
  uint32_t flow = 5000000;  // flow rate 5 mL/min (units is in nL/min)
  uint8_t length = 5 + 4;   // minimum length is 5 (len + cmd + dev
                            // + 2 CRC), + 4 byte argument
  int i;
  // compose the basic message, break the flow argument into 4 bytes
  uint8_t msg[8] = {addr,length,cmd,dev,(flow>>24)&0xFF, (flow>>16)&0xFF,
                (flow>>8)&0xFF, flow&0xFF};
  for (i=0; i<8; i++)       // transfer the msg bytes into the tx_buffer
    tx_buffer[i] = msg[i];
  AppendCrc(tx_buffer);
}
```

Reading Responses from the Board

The examples above do not show how to read messages from the board. For I²C you would send the read address [shift base address left 1 bit, then set bit 0, ((addr<<1) | 1)]. Then continue to read bytes (clocking the SCL line) until you have received the expected number of bytes in the response.

When reading a UART response you would need to convert the ASCII encoded packet bytes back into binary and then interpret the meaning of the response. Here is a code sample that would convert two hexadecimal characters into the equivalent binary value.

```
uint8_t hex2bin(uint8_t c1, uint8_t c2)
{
    uint8_t b;            // this will be the resulting binary byte

    if (c1 < 'A')       // if first char is '0' through '9'
        b = c1 - '0'; // convert to binary value 0 to 9
    else                  // otherwise it is 'A' through 'F'
        b = c1 - 'A' + 10;  // convert to binary value 10 to 15
    b = b << 4;           // put result in the upper 4 bits

    if (c2 < 'A')       // do the same with second char
        b += c1 - '0';    // and add it to the lower 4 bits
    else
        b += c1 - 'A' + 10;

    return b;
}
```