Getting started with the I2C parallel master module

Revision 13

1 Module characteristics

1.1 Characteristics

- Powered with 3.3V or 5V
- Two LEDs (power + activity)
- Two modes of operation:
    - Printer mode
    - IO mode

1.2 Power

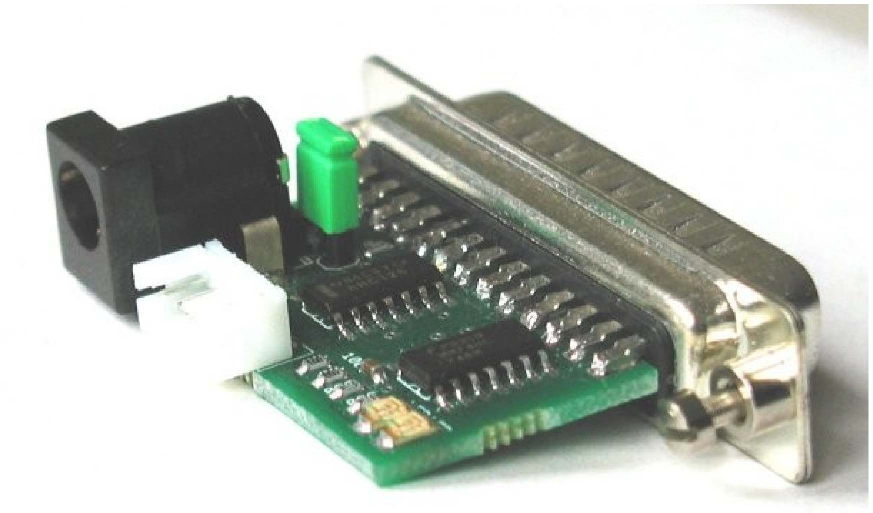
The module is powered using a 2.1mm jack. Many DC power wall adapters have this type of jack.
The jack needs to be center-positive (ground on the outside, power in the inside), and provide a DC 3.3V or 5V regulated power.

If you have a high voltage DC (like 9V or 12V), a cable like this one is adequate to create a 5V supply.
http://www.knjn.com/ShopItemsPics/DCcable5V.jpg
http://www.knjn.com/ShopCablesPower.html

1.3 C compiler

C code is provided to control the module.

A suitable C compiler is Microsoft Visual C++ 5.0 or 6.0, or any other Win32 compiler like lcc-win32 or Digital Mars



2 Modes of operation

2.1 Modes

The I2C parallel master can work in two modes:

1. Printer mode (jumper off)
2. I/O mode (jumper on)

2.1.1 I2C parallel master in printer mode

In this mode, the I2C parallel master appears as a printer to the PC. As a consequence, no driver is necessary but only I2C write commands can be issued, and only one master is supported on the I2C bus.

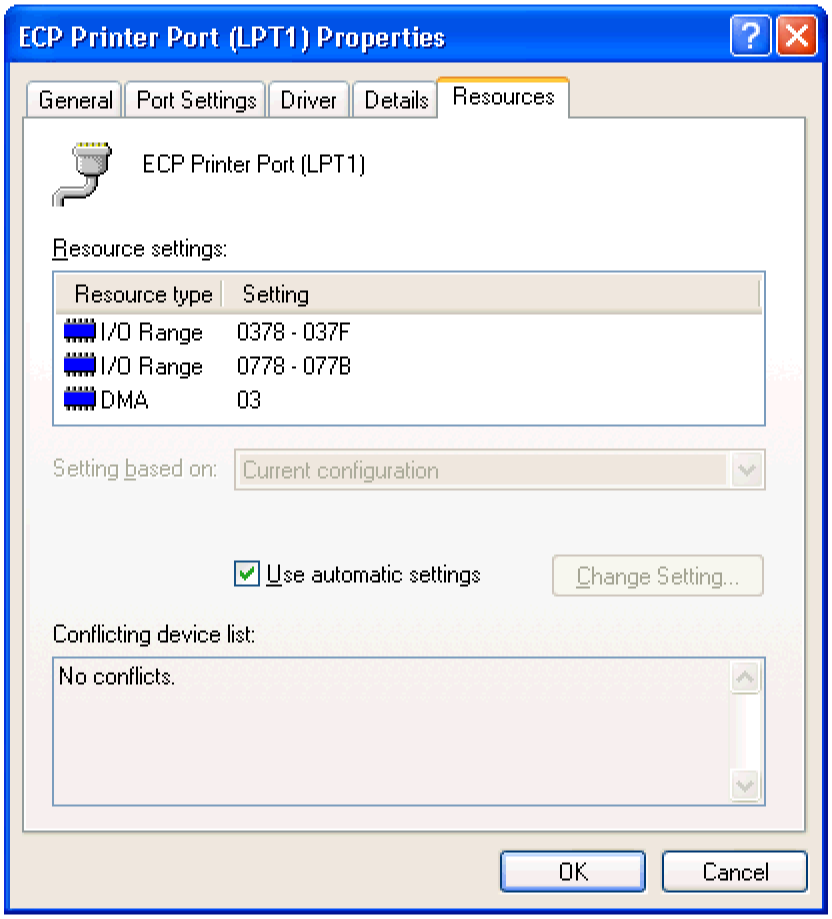
See the “I2C_PARALLEL_PRINTER.c” file.

Note: in printer mode, remove the jumper from the I2C parallel master module.

2.1.2 I2C parallel master in I/O mode

In this mode, the I2C parallel master is controlled directly by issuing IO commands to the PC’s parallel port. That gives more control over the port and allows I2C read and writes.

See the “I2C_PARALLEL_IO.c” file. Remember to update the code with the address of your parallel port. The most common value is 0x378.



ECP Printer Port (LPT1)

Resource settings:

Resource type	Setting
I/O Range	0378 - 037F
I/O Range	0778 - 077B
DMA	03

Setting based on: Current configuration

[✓] Use automatic settings

Change Setting...

Conflicting device list:

No conflicts.

Notes:

- In IO mode, the jumper needs to be present on the I2C parallel master module if you issue any I2C reads. Without the jumper, the board always returns 0xFF on reads.
- Under Windows 2000 and Windows XP, a special driver needs to be installed to allow IO commands to be issued to the parallel port – see the “UserPort.zip” file.

3 Printer mode

In printer mode, the module fakes to be a printer to Windows. No Windows driver is required, but only I2C write commands can be issued.

3.1 Example code

```
char* LPTname = "LPT1";                         // parallel port used

// Parallel I2C bits
#define SCL 0x02
#define SDA 0x10

FILE* F;
BYTE contrast = 0;

void BeginTransaction()
{
    F = fopen(LPTname, "wb");
}

void OpenI2Cport()
{
    BeginTransaction();
}

void EndTransaction()
{
    fclose(F);
}

void CloseI2Cport()
{
    EndTransaction();
}

void SendFlush()
{
    //fflush(F);

    // flush doesn't work, close and re-open the connection...
    EndTransaction();
    BeginTransaction();
}

void TransactionSend(BYTE vbo)
{
    fputc(~vbo, F);
}

void Parallel_I2C_start()
{
    TransactionSend(SCL | SDA);
    TransactionSend(SCL);
    TransactionSend(0);
}

void Parallel_I2C_stop()
{
    TransactionSend(0);
    TransactionSend(SCL);
    TransactionSend(SCL | SDA);
}

void Parallel_I2C_sendbit(BYTE v)
{
    v = (v & 1) ? SDA : 0;
    TransactionSend(v);
    TransactionSend((BYTE)(v | SCL));
    TransactionSend(v);
}

void Parallel_I2C_sendbyte(BYTE b)
{
    int i;
```

```
	for(i=7; i>=0; i--) Parallel_I2C_sendbit((BYTE)(b >> i));
	Parallel_I2C_sendbit(1);  // ACK
}

void Parallel_I2C_sendbuf(char* buf, int len)
{
	int i;
	for(i=0; i<len; i++) Parallel_I2C_sendbyte(buf[i]);
}

void Parallel_I2C_Write(BYTE I2C_address, char* buf, int len)
{
	Parallel_I2C_start();
	Parallel_I2C_sendbyte(I2C_address);
	Parallel_I2C_sendbuf(buf, len);
	Parallel_I2C_stop();
}
```

4 IO mode

In IO mode, full control of the module is achieved. I2C reads and writes can be issued.

4.1 Windows driver

With Windows 2000 & XP, the PC requires a driver to allow IO accesses.

We recommend using a driver like UserPort. Get it here
http://www.writelog.com/support/lpt_port_support_on_windows_nt.htm

4.2 Example code

```
// make sure this match the address of the parallel port of your PC
#define    LPT_PORT            0x378

// Parallel I2C bits
#define SCL 0x02
#define SDA 0x10

BYTE contrast = 0;

void BeginTransaction()
{
    // UserPort doesn't require any code

    // if you use GiveIo, uncomment this:
/*
    HANDLE h = CreateFile("\\\\.\\giveio", GENERIC_READ, 0, 0, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);
    assert(h!=INVALID_HANDLE_VALUE);
    CloseHandle(h); // we can close right away, since GiveIO's effect remains
*/
}

void OpenI2Cport()
{
    BeginTransaction();
}

void EndTransaction()
{
}

void CloseI2Cport()
{
    EndTransaction();
}

void SendFlush()
{
}

void IOwrite(WORD adr, BYTE data)
{
    _asm
    {
            mov dx, adr
            mov al, data
            out dx, al
    }
}

BYTE IOread(WORD adr)
{
    _asm
    {
            mov dx, adr
            in al, dx
    }
}

void TransactionSend(BYTE vbo)
{
    int i;
    for(i=0; i<3; i++) IOwrite(LPT_PORT+0, (BYTE)~vbo);      // emulate a delay by doing the IO multiple times
    IOwrite(LPT_PORT+2, 0x0D);  // strobe
    IOwrite(LPT_PORT+2, 0x0C);
}

void Parallel_I2C_start()
{
    TransactionSend(SCL | SDA);
    TransactionSend(SCL);
    TransactionSend(0);
}

void Parallel_I2C_stop()
{
    TransactionSend(0);
    TransactionSend(SCL);
    TransactionSend(SCL | SDA);
}

void Parallel_I2C_writebit(BYTE v)
{
    v = (v & 1) ? SDA : 0;
    TransactionSend(v);
    TransactionSend((BYTE)(v | SCL));
    while(((IOread(LPT_PORT+1) >> 6) & 0x01)==0x00) Sleep(1);      // wait for slow peripherals that may pull SCL low
    TransactionSend(v);
}

BYTE Parallel_I2C_readbit()
{
    BYTE result;

    TransactionSend(SDA);
    TransactionSend(SDA | SCL);
    while(((IOread(LPT_PORT+1) >> 6) & 0x01)==0x00) Sleep(1);      // wait for slow peripherals that may pull SCL low
    result = (IOread(LPT_PORT+1) >> 4) & 0x01;
    TransactionSend(SDA);

    return result;
}

void Parallel_I2C_writebyte(BYTE b)
{
    int i;
    BYTE ACK;

    for(i=7; i>=0; i--) Parallel_I2C_writebit((BYTE)(b >> i));
    ACK = Parallel_I2C_readbit();          // ACK should be 0, as 1 means the device didn't respond
}

BYTE Parallel_I2C_readbyte()
{
    int i;
    BYTE ACK, result = 0;

    for(i=7; i>=0; i--) result |= Parallel_I2C_readbit() << i;
    ACK = Parallel_I2C_readbit();          // ACK should be 0, as 1 means the device didn't respond

    return result;
}

void Parallel_I2C_writebuf(char* buf, int len)
{
    int i;
    for(i=0; i<len; i++) Parallel_I2C_writebyte(buf[i]);
}

void Parallel_I2C_readbuf(char* buf, int len)
{
    int i;
    for(i=0; i<len; i++) buf[i] = Parallel_I2C_readbyte();
}

void Parallel_I2C_Write(BYTE I2C_address, char* buf, int len)
{
    assert((I2C_address & 0x01)==0x00);             // make sure the address is even

    Parallel_I2C_start();
    Parallel_I2C_writebyte(I2C_address);
    Parallel_I2C_writebuf(buf, len);
    Parallel_I2C_stop();
}

void Parallel_I2C_Read(BYTE I2C_address, char* buf, int len)
{
    assert((I2C_address & 0x01)==0x01);          // make sure the address is odd

    Parallel_I2C_start();
    Parallel_I2C_writebyte(I2C_address);
    Parallel_I2C_readbuf(buf, len);
    Parallel_I2C_stop();
}
```