

# LED Reference Design Application Note

---

(VERSION 1.2)



2009/06/24

*Authors:*

XMOS LTD.

Copyright © 2009, XMOS Ltd.  
All Rights Reserved

## Contents

<b>1 Summary</b>	<b>4</b>
1.1 XC-3 Scan Board . . . . .	6
<b>2 LED Reference Design Specification</b>	<b>7</b>
<b>3 XS1 Architecture</b>	<b>7</b>
3.0.1 XMOS XS1-G Programmable Devices . . . . .	8
3.0.2 Software Libraries . . . . .	8
3.0.3 XC Language . . . . .	9
<b>4 Software Design</b>	<b>9</b>
4.1 Ethernet MII . . . . .	11
4.2 Layer 2 Ethernet Switch . . . . .	11
4.3 Internal Ethernet Server . . . . .	12
4.4 Data Processing . . . . .	13
4.5 LED Driving . . . . .	14
4.6 Thread Diagram . . . . .	15
4.7 Thread Descriptions . . . . .	15
<b>5 Firmware Control</b>	<b>34</b>
5.1 Bootloader . . . . .	34
5.2 Upgrading . . . . .	34
5.3 Future Implementation . . . . .	35
<b>6 LED Reference Design Configuration Application</b>	<b>35</b>

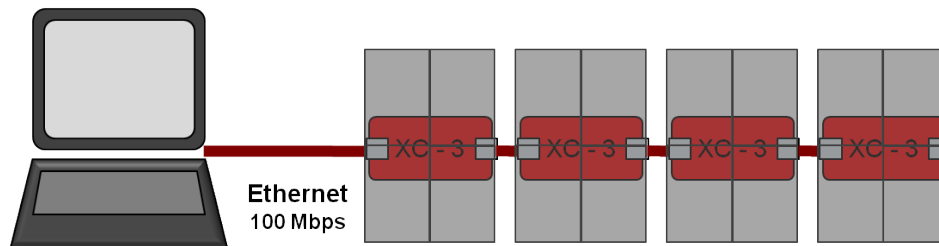
---

6.1	Arguments	35
6.2	Commands	36
6.3	Persistence	37
<b>7</b>	<b>Mplayer plugin</b>	<b>37</b>
7.1	Chain Specification	38
7.2	Xudp Arguments	38
7.3	Example Setup	39
<b>8</b>	<b>Data Protocol</b>	<b>40</b>
8.1	Packet Formats	40
8.1.1	Header	40
8.1.2	Data	41
8.1.3	Latch	42
8.1.4	Gamma Adjust	42
8.1.5	Intensity Adjust	42
8.1.6	Soft Intensity Adjust	43
8.1.7	Soft Intensity Adjust to Pixel	43
8.1.8	Reset	43
8.1.9	Autoconfiguration	44
8.1.10	Change Driver Type	44
<b>9</b>	<b>Autoconfiguration</b>	<b>44</b>
9.1	Packet Types	45
9.2	Autoconfiguration Method	47

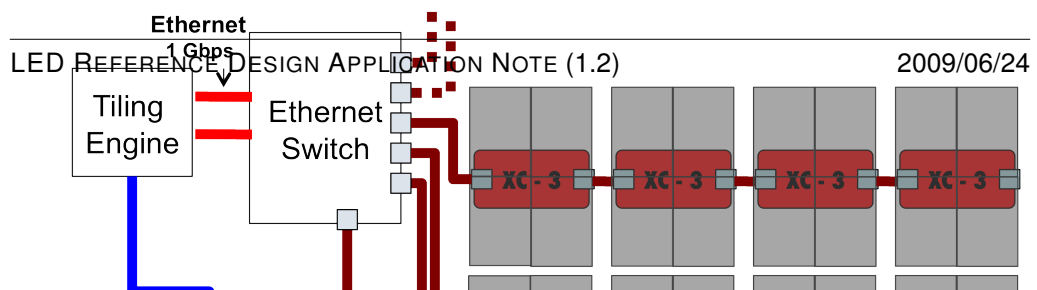
<b>10 Memory Requirements</b>	<b>48</b>
10.1 Total Required Memory . . . . .	48
10.2 Detailed Required Memory . . . . .	48
10.3 Key Buffer Locations . . . . .	49
10.4 Key Look-up Table Locations . . . . .	49
<b>11 Software Structure</b>	<b>49</b>
<b>12 Related Documents</b>	<b>51</b>

## 1 Summary

The XMOS LED Reference Design is an Ethernet-based system featuring daisy-chains of 100BASE-T scan boards off a central 100/1000BASE-T Ethernet switch. The design uses standard Ethernet technologies, where possible, to enable ease of upgradability, component availability and compliance with third-party technologies. The XMOS solution is based on the XS1-G4 software defined silicon programmable device.



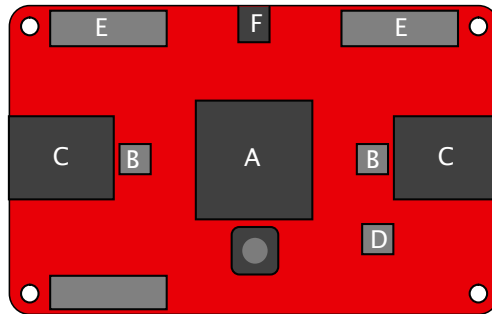
The reference design can be used out-of-the-box as a demonstration or included in a complete video wall system. For demonstrations, a PC (or Host) can be directly connected to the Ethernet daisychain, and a standard video player such as the open-source *mplayer* application can be used to broadcast video; see the *Mplayer Plugin* section (7) for more details.



For large video wall systems, a separate distribution or tiling unit is required. This inputs DVI, HDMI or VGA video at HD resolutions, separates the data for each of the tiles in the system, packetises it and distributes over one or many gigabit Ethernet streams. A standard 100+1000 Ethernet switch can be used to break the streams over multiple 100Mbps daisychains. Any spare ports on the switch can be routed back to the host or to a diagnostic port for technician access.

### 1.1 XC-3 Scan Board

The LED Reference Design is based on the XC-3 scan board, which provides the following core components:



A XS1-G4 Device in 144BGA package

B 2 x SMSC LAN8700 10/100BASE-T Ethernet Physical layer device in 36pin QFN

C 2 x Halo RJ45 Connector

D Amtel AT25DF041A 4Mbit SPI flash

E 2 x 16-way 2mm IDC headers

F 5 volt to 3.3/1.0 volt dual regulator

## 2 LED Reference Design Specification

<b>Functionality</b>	
Provides ethernet interface, ethernet switch, image processing and LED driving for LED control applications.	
<b>Supported Devices</b>	
XMOS Devices	XC-3 LED Tile Controller Card
<b>Resource Usage</b>	
Thread Usage	14
Memory Usage	80Kbytes, application dependent
Required Ports	MII: 15×1-bit, 4×4-bit LED: 10×1-bit, 1×4-bit, 1×8-bit
<b>Requirements</b>	
Required Development Tools	XMOS Desktop Tools v9.5.0 or later
<b>Licensing and Support</b>	
Reference code provided free of charge to XMOS customers for unrestricted use on XMOS devices only.	
Reference code is maintained by XMOS Limited.	

## 3 XS1 Architecture

The XS1 architecture consists of one or more processing cores, called XCores™, which have the following properties:

- Each XCore can execute up to eight threads concurrently, at a speed of up to 400 MIPS. Each thread has a dedicated register set enabling it to operate as a logical core.
- The eight threads share a single 64 KByte unified memory with no access collisions.

- Integer and fixed point operations are provided for efficient DSP and cryptographic operations.
- 64 I/O general purpose pins are provided, which can be programmed from software. Thread execution is deterministic and hence each thread can implement a hard real-time I/O task, regardless of the behaviour of other threads.
- I/O pins are grouped into logical ports of width 1, 4, 8, 16 and 32 bits. Each port incorporates serialisation/deserialisation, synchronisation with the external interface and precision timing.
- Each XCore incorporates eight timers that measure time relative to a 100 MHz reference clock.

The architecture is designed to support standard programming languages such as C. Extensions to standard languages, libraries, or the use of assembly language provide access to the full benefits of the instruction set.

### **3.0.1 XMOS XS1-G Programmable Devices**

The XS1-G family includes the XS1-G4 device, which integrates four XCore processors. Each device is connected to a high performance switch interconnect via four internal links. Each link is capable of transferring data at 800 Mbits/second. The switch provides full connectivity between the cores on the programmable device, and also provides up to sixteen external links. Each external link is capable of transferring data at up to 400 Mbits/second.

Other members of the family include the XS1-G1 and XS1-G2. XS1 devices can be used standalone, or can be connected together using links to create a network of XCore processors.

### **3.0.2 Software Libraries**

The XS1 architecture implements hardware as software, so that hardware solutions are just software libraries. These software solutions are compiled and linked like ordinary C code into a binary file which can be loaded and executed on the device.

The LED Reference Design is a library of source files that can be included in a larger application. The application can then be compiled and deployed onto a device to provide your LED solution.

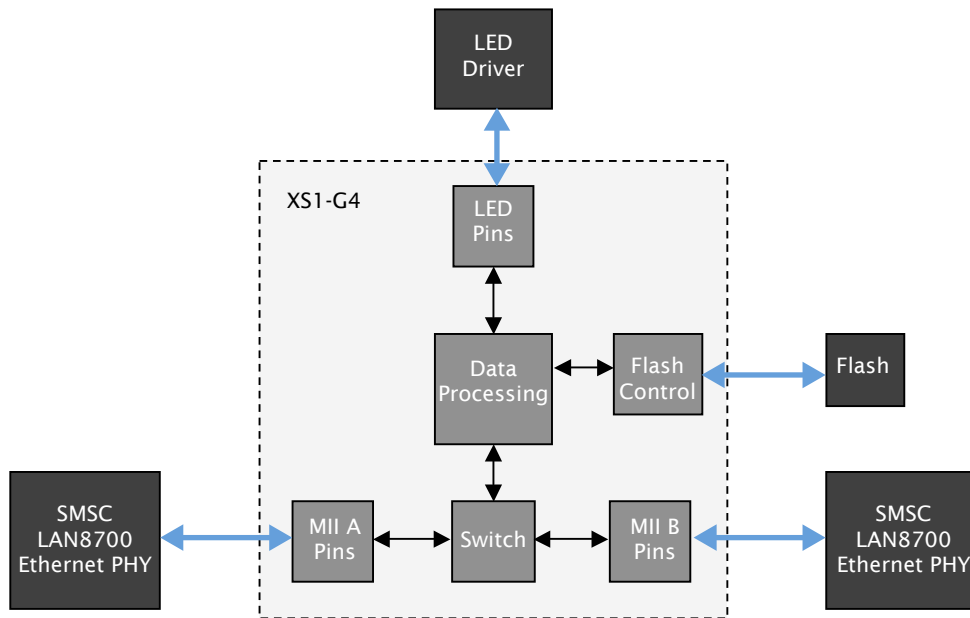
### **3.0.3 XC Language**

The XMOS originated XC language is based upon C. XC is a concurrent and realtime programming language designed to target the XS1-G architecture. XC programs are easy to write and debug—free from deadlocks, race conditions and memory violations—and can be compiled to produce high performance multicore designs.

XC uses channels to provide high-speed bidirectional communication and synchronisation between channel ends within threads on the same processor, a different processor on the same chip or a different chip.

## **4 Software Design**

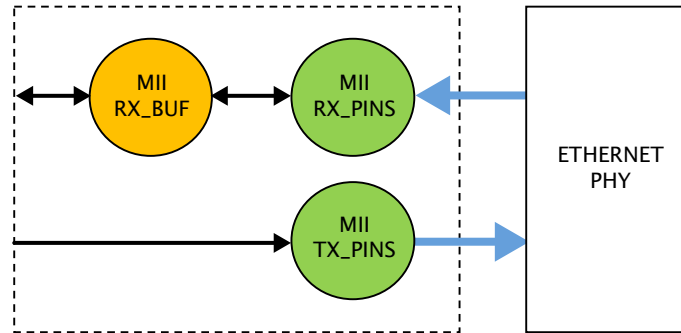
The table below shows a top-level overview of the LED Reference Design program.



The reference design incorporates the following Ethernet capabilities.

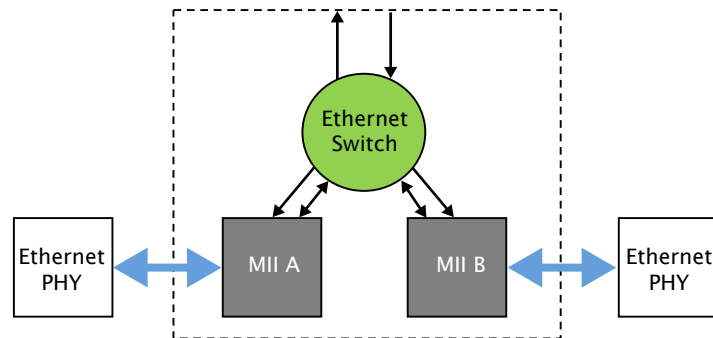
- Two 10/100Mbps full duplex MII layers
- Dynamic Receive buffers (FIFO)
- 3-port Layer 2 Ethernet Switch
- Internal UDP/IP Ethernet server

### 4.1 Ethernet MII



The application uses the standard XMOS Media Independent Interface (MII) layers, which are available to download from <http://www.xmos.com>. The layers feature low-level receive and transmit threads, which also handle cyclic redundancy checking (CRC). A separate buffering thread is used for receipt of data packets. For packet transmission, the transmit thread is buffered, limiting transmit rates to around 95Mbps, and assuming the source of data has the packet entirely ready for transmission.

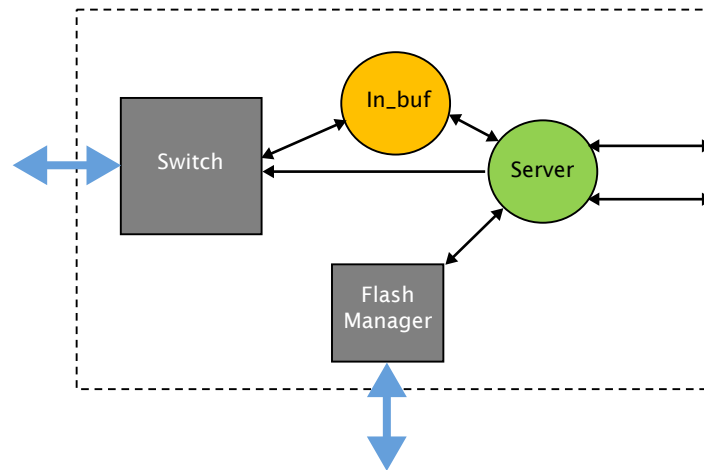
### 4.2 Layer 2 Ethernet Switch



A simple Ethernet switch is implemented in software. Ports have a single internal node, so only one local MAC address is stored, rather than using large MAC tables. When one of the receive buffers indicates a packet is ready, it is read from the receive buffer thread into a local buffer. Then the decision is made if

this packet is for the local node (MAC matches), a remote node (MAC does not match), or both (broadcast). Packets for the local node are forwarded to the local server. Packets for remote nodes are passed to the buffer of the other transmit thread. The TTL for IP packets are adjusted.

### 4.3 Internal Ethernet Server



The internal Ethernet server supports the following low-level protocols.

- Internet Protocol (IP), partial implementation
  - The IP server defaults to IP address **192.168.0.254** unless auto-configured.
- User Datagram Protocol (UDP) over IP

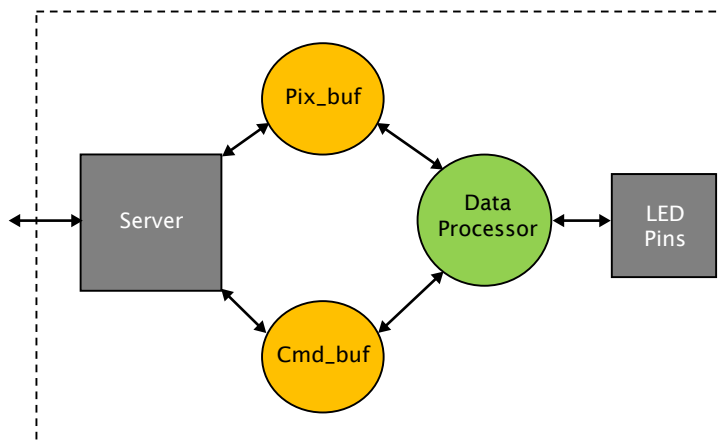
The server also supports the following high-level protocols

- Internet Control Message Protocol (ICMP)
- Address Resolution Protocol (ARP)
- Trivial File Transfer Protocol (TFTP)

- TFTP server accepts only "PUT" of files with filenames of "firmware" or "reset"
  - See the Firmware Control upgrade section (5) for more details
- XMOS Video and Configuration Data
  - See Data Protocol section (8) for more details
- XMOS IP Autoconfiguration
  - See Autoconfiguration section (9) for more details.

The server stores and retrieves firmware data and boot images in the SPI flash through a channel to the flash manager.

#### 4.4 Data Processing

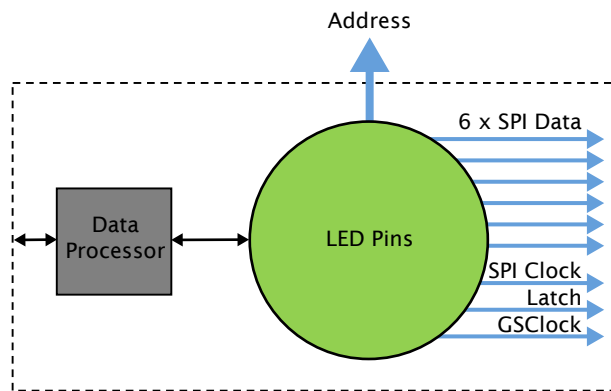


Pixel data and commands such as gamma configuration are buffered separately before being passed to the data processing thread. The pixel buffer supports a double-buffering scheme, where the server writes into one buffer and the processor reads from the other, and preventing tearing. The data processor supports the following adjustments:

- Intensity Adjustment
  - For each of the three colour channels, an 8-bit intensity modifier is stored
  - Used for simple testing
- Gamma adjustment
  - For each of the three colour channels, an 8-bit to 16-bit look-up table is stored
  - This is not required to be a standard gamma curve, as it is generated from the Ethernet host

See the *LED Reference Design Configuration Application* section (6) for more details.

## 4.5 LED Driving

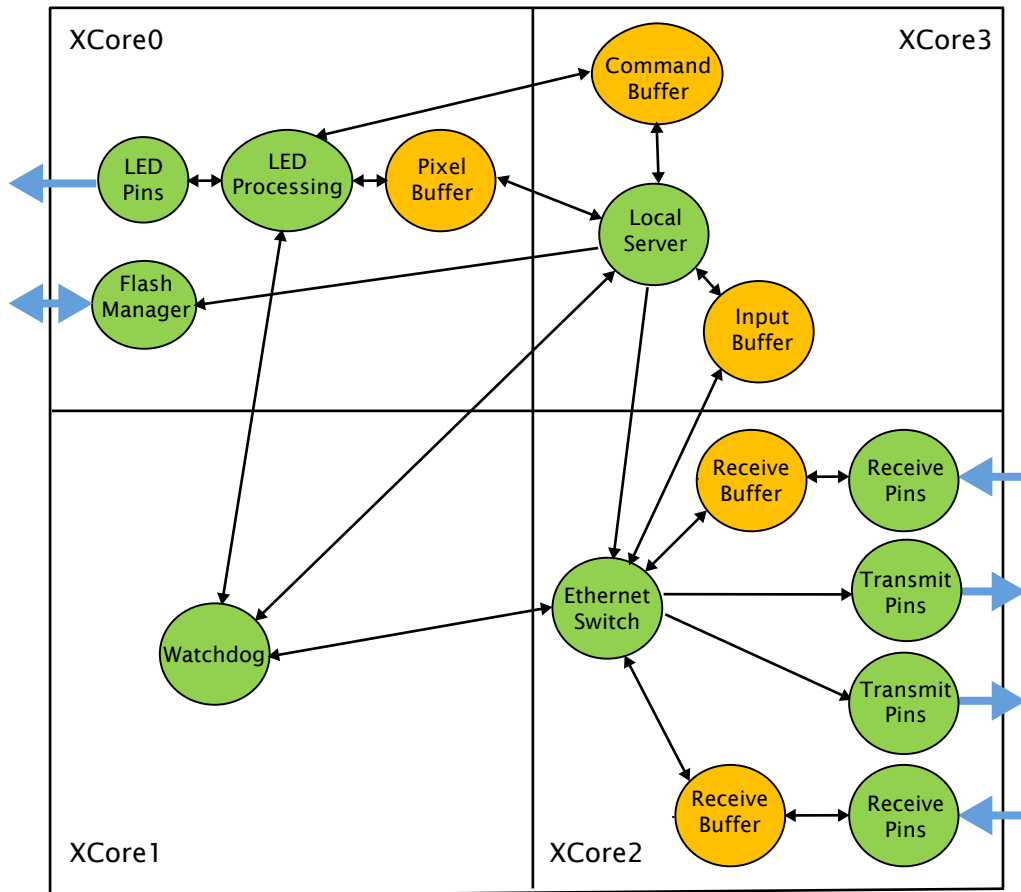


SPI Data is driven out the low-level LED pins thread through six parallel SPI data lines. This is clocked out with a shared SPI clock at 25Mhz, with separate latch and grayscale-clock signals. This thread also manages the address lines to control the scan rate of multiplexed LED modules. As well as supplying pixel data, it also supports register writes for current gain adjustment.

This thread does not currently implement driver feedback.

### 4.6 Thread Diagram

The diagram below shows the complete thread structure for the XMOS LED Reference Design.



### 4.7 Thread Descriptions

This section provides a brief overview of the function of each thread used in the LED Reference Design, and details of the channels and ports it uses.

<b>MII_RX_PINS</b>		
Description	Physical MII receive thread. Physical layer thread which interfaces with the 10/100Mbps Media Independent Interface from the Ethernet Phy to the XCore. Supports line-error detection as well as running Cyclic Redundancy Checks on the Ethernet frame.	
Channels	c_mii_rx	Streaming unidirectional output
Port	p_mii_rxd	Buffered input port, 32bit Transfer Width , 4bit Port Width
	p_mii_rxdv	Input port, 1bit Port Width
	p_mii_rxer	Input port, 1bit Port Width

### Example Code

```
// Start of frame
p_mii_rxdv when pinseq(1) :> int hi;
p_mii_rxd when pinseq(0xD) :> int sof;

tmr :> time;

p_mii_rxd :> word;
outuint(c_mii_rx, FRAME_DATA);
outuint(c_mii_rx, word);
length+=4;
crc32(crc, word, poly);
p_mii_rxd :> word;
outuint(c_mii_rx, FRAME_DATA);
outuint(c_mii_rx, word);
length+=4;
crc32(crc, word, poly);
```

<b>MII_RX_BUF</b>		
Description	MII receive buffering thread. Intermediary buffering thread which supports simultaneous sinking of frame data from <code>mii_rx_pins</code> , and sourcing of the previous frame data to a client thread. Uses a circular FIFO buffer.	
Channels	<code>c_mii_data</code>	Streaming unidirectional input
	<code>c_mii_client</code>	Streaming bidirectional
Ports	None	

### Example Code

```

select
{
  case inuint_byref(c_mii_data, x):
  {
    if (x != FRAME\_DATA)
    {
      unsigned int nbytes = inuint(c_mii_data);
      if (overflow || (nbytes==0))
      {
        outIndex = startIndex + 1;
        overflow = 0;
      }
      else
      {
        buffer[startIndex] = nbytes;
        startIndex = outIndex;
        outIndex++;
        isEmpty = 0;
      }
    }
  }
  else
  {
    x = inuint(c_mii_data);
    buffer[outIndex] = x;
  }
}

```

```
        outIndex += 1;
        // assume size is power of 2
        outIndex &= (MII_RX_FIFO_SIZE) - 1;
        overFlow |= (outIndex == inIndex);
        if (overFlow)
        {
            outIndex = startIndex + 1;
        }
    }
    break;
}
case (!isEmpty) => inuint_byref(c_mii_client, x):
    outuint(c_mii_client, buffer[inIndex]);
    inIndex += 1;
    inIndex &= (MII_RX_FIFO_SIZE) - 1;
    isEmpty = (inIndex == startIndex);
    break;
}
```

<b>MII_TX_PINS</b>		
Description	Physical MII transmit thread. Physical layer thread which interfaces with the 10/100Mbps Media Independent Interface from the Xcore to the Ethernet Phy. Contains one Ethernet-frame buffer. Calculates Cyclic Redundancy Check value on the Ethernet frame.	
Channels	c_mii_tx	Streaming unidirectional input
Ports	p_mii_txd	Buffered output port, 32bit Transfer Width , 4bit Port Width
Ports	None	

### Example Code

```
// Preamble
p_mii_txd <: 0x55555555;
p_mii_txd <: 0x55555555;
p_mii_txd <: 0xD5555555;
tmr :> time;

word = buf[index++];
p_mii_txd <: word;
crc32(crc, ~word, poly);
bytes_left -=4;

while (bytes_left > 3)
{
    word = buf[index++];
    p_mii_txd <: word;
    crc32(crc, word, poly);
    bytes_left -= 4;
}
```

<b>ETHERNET SWITCH</b>		
Description	Layer 2 Software Ethernet Switch. A software Ethernet switch, sinking frame data from either of the two external Ethernet interfaces or the one local interface. Uses store-and-forward architecture to pass frame onwards. Detects local MAC address from frame data.	
Channels	cExtRx0, cExtRx1, cLocRx	Streaming bidirectional
	cExtTx0, cExtTx1, cLocTx	Streaming bidirectional
Ports	None	

### Example Code

```
// Received data from external receiver 1
case inuint_byref(cExtRx1, numbytes):
// Retrieve and validate packet
  if (ethPhyRx(cExtRx1, packet, numbytes, timestamp) == 0)
  {
  // Do MAC comparison
    if ((getWord(packet.pdata[0]) == mymacaddress[0]) &&
        ((getWord(packet.pdata[1]) & 0xFFFF0000) ==
         mymacaddress[1]))
    {
      // Packet is destined for local node
      getMac(mymacaddress, cLocTx);
      ethPhyTx(cLocTx, packet, timestamp, 2);
    }
    else if ((getWord(packet.pdata[0]) ==
              broadcastaddress[0]) &&
             ((getWord(packet.pdata[1]) & 0xFFFF0000) ==
              broadcastaddress[1]))
    {
      // Packet is broadcast
      ethPhyTx(cExtTx0, packet, timestamp, 0);
      getMac(mymacaddress, cLocTx);
    }
  }
}
```

---

```
        ethPhyTx(cLocTx, packet, timestamp, 2);
    }
else
{
    // Packet is neither broadcast or local,
    // forward onwards
    ethPhyTx(cExtTx0, packet, timestamp, 0);
}
}
```

<b>LOCAL SERVER</b>		
Description	Local Ethernet server. This thread deals with sorting and parsing of all local Ethernet frames. It responds to ICMP, ARP, parses LED video data and latch commands sent over UDP. LED specific data is forwarded to the next thread. Future implementations will include BOOTP/DHCP and TFTP for boot-from-Ethernet.	
Channels	cRx	streaming bidirectional Data receive
	cTx	streaming bidirectional Data receive
	cLedData	Streaming bidirectional Pixel data output
	cLedCmd	Streaming bidirectional Command output
Ports	None	

### Example Code

```
// ICMP Packets
if (getShort(m->ethertype) == ETHERTYPE_IP &&
    getChar(i->proto) == PROTO_ICMP)
{
    s_packetIp *i;
    s_packetIcmp *icmp;

    i = (s_packetIp *)m->payload;
    icmp = (s_packetIcmp *)i->payload;

    // Check if targetting our IP address
    if (memcmp(i->dest, addresses->ipAddress, 4) == 0)
    {
        // Check if valid IP echo request
        if (icmp->type == ICMP_ECHOREQUEST &&
            icmp->code == ICMP_CODE)
        {
            unsigned short *ptr;
            unsigned icmpchecksum=0;
            int j;
```

```
// Create reply
icmp->type = ICMP_ECHOREPLY;

// Recalculate ICMP checksum
icmp->checksum = 0;
ptr = (unsigned short *)&icmp->type;
for (j=0; j < 20; j++)
{
    icmpchecksum += getShort(ptr[j]);
}
icmp->checksum = ~getShort((icmpchecksum &
    0xFFFF) + (icmpchecksum >> 16));

// Reverse destination and source MAC addresses
memswap(m->destmac, m->sourcemac, 6);

// Reverse destination and source IP addresses
memswap(i->dest, i->source , 4);

// Transmit packet in direction it came from
ethPhyTx(cTx, packet, &null, direction + 1);
}
}
}
```

<b>DATA BUFFER</b>		
Description	Double-buffered frame store. Frame buffer for storing pixel data. Sinks incoming data from the local server, and sources data to the LED driving threads. Supports frame turnaround without tearing.	
Channels	cIn	Streaming bidirectional pixel sink
	cOut	Streaming bidirectional pixel source
Ports	None	

### Example Code

```
// Source dump
// Guard prevents more data pushed in after frame switch
case (bufswaptriggerN) => inuint_byref(cIn, pixelptr):
    if (pixelptr == -1)
    {
        // End of frame signal from source
        // Frame is not swapped until sink also completes
        bufswaptriggerN = 0;
    }
    else
    {
        // New data from source
        int len = inuint(cIn);
        pixelptr += inbufptr;
        while (len > 0)
        {
            buffer[pixelptr] = inuint(cIn);
            pixelptr++;
            len-=3;
        }
    }
    break;
```

<b>COMMAND BUFFER</b>		
Description	Buffering FIFO for commands. Sinks commands such as gamma LUT changes, intensity value changes from the local server and buffers them on a packet-by-packet basis in a circular fifo for the LED driving thread to receive when ready.	
Channels	cln	Streaming bidirectional command sink
	cOut	Streaming bidirectional command source
Ports	None	

### Example Code

```
// Packet request from sink
case inuint_byref(cOut, temp):
    // Check if FIFO is empty
    if (wptr == rptr)
    {
        // Nack
        outuint(cOut, 1);
    }
    else
    {
        // Retrieve packet size from buffer
        unsigned int num = buffer[rptr++];
        rptr &= BUFFERMASK;
        // Ack
        outuint(cOut, 0);
        // Send packet size
        outuint(cOut, num);
        while (num)
        {
            num--;
            outuint(cOut, buffer[rptr++]);
            rptr &= BUFFERMASK;
        }
    }
}
```

<b>COLOUR PROCESSING</b>		
Description	Colour processing of pixel data. Receives pixel data and commands from the buffers and applies Gamma and intensity correction.	
Channels	cLedData	Streaming bidirectional pixel input
	cLedCmd	Streaming bidirectional command input
	cOut	Streaming bidirectional pixel output
Ports	None	

### Example Code

```
int ledprocess_led(int x, int y, int c, int data)
{
    int retdata;

    #ifdef GAMMA_CORRECT
        retdata = bitrev(gammaLUT[2-c][data & 0xFF]) >> 16;
    #else
        retdata = (data & 0xFF) * (data & 0xFF);
    #endif
    retdata = (retdata * (intensityadjust[2 - c]+1) *
        (pixintensity[y][x][2 - c]+1)) >> 16;

    return (retdata);
}
```

<b>LED REFORMAT</b>		
Description	Pixel data reformat. Sinks data from frame buffer and reformats it into a 6-bit wide SPI for output on a 8-bit port by the driving thread. Uses a Morton LUT for bit expansion and interleaving.	
Channels	cln	Streaming bidirectional pixel sink
	cOut	Streaming bidirectional SPI data source
Ports	None	

### Example Code

```
// Pass that data, reformatted, into the LED drivers
for (channel=0; channel < 16; channel++)
{
  for (drivernum=0;
      drivernum < BUFFER_SIZE >> 4; drivernum++)
  {
    // Each loop corresponds to two words
    // of data for two pixels
    unsigned d0, d1, d2, d3, d4, d5;
    unsigned i = (15 - channel) +
      (((BUFFER_SIZE >> 4) - 1 - drivernum) << 4);

    d0 = r1[i]; d1 = g1[i]; d2=b1[i];
    d3 = r2[i]; d4 = g2[i]; d5=b2[i];
    j=4;
    while (j)
    {
      unsigned int outputval=0;
      doMorton(outputval, d5);
      doMorton(outputval, d4);
      doMorton(outputval, d3);
      doMorton(outputval, d2);
      doMorton(outputval, d1);
      doMorton(outputval, d0);
    }
  }
}
```

---

```
        // outputval contains 4 bits for
        // each channel -> 4x8 bits
        outuint(cOut, outputval);
        j--;
    }
}
}
```

<b>LED PIN DRIVER</b>		
Description	Physical SPI interface. Sinks data from frame buffer, reformats and outputs on the physical SPI interface. Controls latching and clock generation as well as register writes.	
Channels	cln	Streaming bidirectional pixel/command input
Port	p_spi_r0	Buffered output port, 32bit Transfer Width, 1bit Port Width
	p_spi_g0	Buffered output port, 32bit Transfer Width, 1bit Port Width
	p_spi_b1	Buffered output port, 32bit Transfer Width, 1bit Port Width
	p_spi_r1	Buffered output port, 32bit Transfer Width, 1bit Port Width
	p_spi_g1	Buffered output port, 32bit Transfer Width, 1bit Port Width
	p_spi_b1	Buffered output port, 32bit Transfer Width, 1bit Port Width
	p_spi_addr	Output port, 4bit Port Width
	p_spi_clk	Buffered output port, 32bit Transfer Width, 1bit Port Width
	p_spi_latch	Buffered output port, 32bit Transfer Width, 1bit Port Width
	p_spi_oe	1bit Port Width

### Example Code

```

for (int channel=0; channel < LEDES_PER_DRIVER;
    channel++)
{
    for (int drivernum=0; drivernum < NUM_DRIVERS;
        drivernum++)
    {
        unsigned ptr = (LEDES_PER_DRIVER -
            (channel + 1)) + ((NUM_DRIVERS -
            (1 + drivernum)) * LEDES_PER_DRIVER);
    }
}

```

```
// Pass new data to the output ports
// NOTE This section changes for different topologies

p_led_out_b0 :16 <: (unsigned)buffers[ptr][0];
p_led_out_g0 :16 <: (unsigned)buffers[ptr][1];
p_led_out_r0 :16 <: (unsigned)buffers[ptr][2];
p_led_out_b1 :16 <: (unsigned)buffers[ptr][3];
p_led_out_g1 :16 <: (unsigned)buffers[ptr][4];
p_led_out_r1 :16 <: (unsigned)buffers[ptr][5];

if (drivernum == (BUFFER_SIZE/LEDS_PER_DRIVER) - 1)
{
  if (channel == LEDS_PER_DRIVER - 1)
  {
    p_spi_ltch :16 <: GLOBL_LATCH;
  }
  else
  {
    p_spi_ltch :16 <: LOCAL_LATCH;
  }
}
else
{
  p_spi_ltch :16 <: 0;
}

// Clock out this data
p_spi_clk <: 0x55555555;
}

// Wait for the latch point
t when timerafter(now + FRAME_TIME) :> now;

// Bring down the latch
```

```
p_spi_ltch :1 <: 0;  
p_spi_clk :2 <: 0x1;  
p_spi_addr <: (unsigned) scanx;
```

<b>SPI FLASH</b>		
Description	SPI Flash read and write. Loads Gamma LUTs off SPI flash and provides auto-update and firmware upgrade capability.	
Channels	cln	bidirectional
Port	p_flash_miso	Buffered input port, 8bit transfer width, 1bit Port Width
	p_flash_mosi	Buffered output port, 8bit transfer width, 1bit Port Width
	p_flash_clk	Buffered output port, 32bit transfer width, 1bit Port Width
	p_flash_ss	Output port, 1bit Port Width

### Example Code

```
void spi_blockerase(int baddr, out port p_flash_ss,
    buffered in port:8 p_flash_miso,
    buffered out port:32 p_flash_clk,
    buffered out port:8 p_flash_mosi)
{
    unsigned null;
    unsigned addr = bitrev(baddr) >> 8;
    spi_write_enable(p_flash_ss, p_flash_miso,
        p_flash_clk, p_flash_mosi);

    p_flash_ss <: 0;
    clockbyte(byterev(bitrev(SPI_BE)), null);
    clock3byte(addr, null);
    p_flash_ss <: 1;
}
```

<b>WATCHDOG</b>		
Description	Thread watchdog. Queries Processor-Switch and System-Switch control registers to check for deadlock and attempt local thread resets or system reset.	
Channels	cWdogSwitch	Unidirectional input
	cWdogServer	Unidirectional input
	cWdogLed	Unidirectional input
Ports	None	

### Example Code

```

case t when timerafter
    (time + WDOG_WAIT/CORECLOCKDIV) :> time:
// Check dog
if (dog_kicked != (1 << NUM_WATCHDOG_CHANS) - 1)
{
    for (int i=0; i<NUM_WATCHDOG_CHANS; i++)
    {
        if (!(dog_kicked & (1<<i)))
        {
            printstr("Watchdog failed on chan ");
            printintln(i);
        }
    }
}
}

```

## 5 Firmware Control

The onboard SPI Flash supports three possible firmwares. It boots the most recent valid image. The local Ethernet server onboard supports automatic upgrading of firmware through TFTP.

### 5.1 Bootloader

On boot, the XCore begins to load its boot image from address 0 of the SPI Flash. This image is a standard boot loader, which runs on only XCore 0. This boot loader runs Cyclic Redundancy Checks (CRC) on all three other possible boot images stored in flash sectors 1,2 and 3, and also checks which image is newest. The best image is then loaded from the SPI Flash, the XB file stored is parsed, and all four cores of the G4 are booted.

This means that a failed or partial upgrade will not corrupt old images.

### 5.2 Upgrading

The local server includes a simple TFTP server, with limited filename support. To make an upgrade use the following command, where *filename.xb* is a standard XMOS binary image, and *tftp* is a standard application included in most versions of Microsoft Windows:

```
tftp --i ipaddress PUT filename.xb firmware
```

XB files can be generated from XE files using the XMOS XOBJDUMP tool. For example use the following command:

```
xobjdump --strip filename.xe
```

On receipt of the TFTP request, the local server makes similar checks to the boot loader. It runs CRC checks on all local images, and also looks for the oldest image. This image is overwritten with the new firmware image and a new timestamp stored.

Finally the system must be reset to run the new firmware. This can either be done with the Configuration Application, or with the following command, where *filename* is a dummy file:

```
*tftp --i /ipaddress/ PUT /filename/ reset
```

### 5.3 Future Implementation

In production systems, the bootloader that is presently stored in SPI Flash, is expected to be stored in the One-Time-Programmable (OTP) memory of the XS1-G4 device. In addition, a secondary emergency boot image would also be stored in OTP. This second image would be loaded by the bootloader should the SPI flash be entirely corrupted. This image should implement a Boot-From-Ethernet mechanism, by providing a MII layer, switch and TFTP server.

## 6 LED Reference Design Configuration Application

The LED Reference Design includes a configuration application that can be used to access individual tiles on the network. The application supports changing colour correction, resetting and configuring tiles.

### 6.1 Arguments

- --prefix=
  - Sets the IP address prefix for nodes on the network
  - Default is **192.168**
- --port=
  - Sets the UDP port for communication to nodes
  - Default is **306**

For example, to communicate with tiles on port 900, and IP addresses 10.0.XXX.XXX, run with arguments:

```
--prefix=10.0 --port=900
```

## 6.2 Commands

- Current Gain Adjustment
  - Syntax: *[intensity/i] ipAddress intensity*
  - Adjusts current gain of the LED drivers; intensity value is an integer between 0 and 255
  - Example: Set full current gain on 192.168.2.3  
**i 2.3 255**
- Software Intensity Adjustment
  - Syntax: *[softintensity/si] [R/G/B/A] ipAddress intensity*
  - Adjusts the intensity multiplier in the XCore software for a particular channel; intensity value is integer between 0 and 255
  - Channel select between **Red,Green,Blue** and **All**
  - Example: Turn off red channel on 192.168.1.1  
**si R 1.1 0**
- Software Intensity Adjustment to Pixel
  - Syntax: *[softintensitypix/sip] [R/G/B/A] ipAddress [x] [y] intensity*
  - Adjusts the intensity multiplier in the XCore software for a particular LED; intensity value is integer between 0 and 255
  - Specify pixel position in x,y coordinates
  - Channel select between **Red,Green,Blue** and **All**
  - Example: Turn off pixel 2,3's red channel on 192.168.1.1  
**sip R 1.1 2 3 0**
- Gamma Adjustment
  - Syntax: *[gamma/g] [R/G/B/A] ipAddress gamma*
  - Writes new gamma LUT (8bit - 16bit) to the node; *gamma* is decimal number between 0.1 and 9.9
  - Result is calculated as:  $gamma\_CHAN[x] = (2^{16} - 1) * (x / (2^8 - 1))^{gamma}$
  - Channel select between **Red,Green,Blue** and **All**
  - Example: Write quadratic curve to all channels on 192.168.9.2  
**g A 9.2 2.0**

- Reset
  - Syntax: *[reset/r] ipAddress*
  - Instruct a node to reset the XCore
  - Example: Reset 192.168.1.1  
**r 1.1**
- Autoconfigure
  - Syntax: *[autoconfigure/ac]*
  - Sends out autoconfiguration packets. Resets any existing configurations. See the Autoconfiguration section (9) for more details
  - Example:  
**ac**
- Change Driver Type
  - Syntax: *[changedriver/cd] [ipAddress] [driverType]*
  - Configure the target node to drive for a different driver type
  - Currently supported values: 1=MB15026, 2=MB15030
  - Example: Change 192.168.1.1 to use MB15030  
**cd 1.1 2**

### 6.3 Persistence

Current gain and software intensity values are non-persistent and reset with a chip reboot. The gamma tables, however, are stored in sector 4 of the onboard SPI Flash and persist with a reboot.

## 7 Mplayer plugin

For demonstration purposes, XMOS has written a plugin for the open-source media player 'MPlayer' (<http://www.mplayerhq.hu>). MPlayer supports most MPEG/VOB, AVI and many other media files. MPlayer can be compiled for Windows, Mac OSX and various Linux distributions. Binaries for Windows 32bit and Mac OSX are included in the release.

The plugin provides an alternative video output driver, **vo\_xudp**, which can be called by giving the command line option:

```
-vo xudp.
```

The purpose of the plugin is to segment the video stream into tiles, packetise the data, and output it over UDP.

## 7.1 Chain Specification

To specify the arrangement of Ethernet chains on the network, arguments must be passed to the plugin which detail this. The format for this is for each chain in IP address order, specify the start of the chain on the screen, and the next point on the screen that this chain runs to. Furthermore, if the chain turns a corner, another point can be specified to continue the chain. Then, the start of the next chain can be specified. See the Example Setup section (7.3) for details.

## 7.2 Xudp Arguments

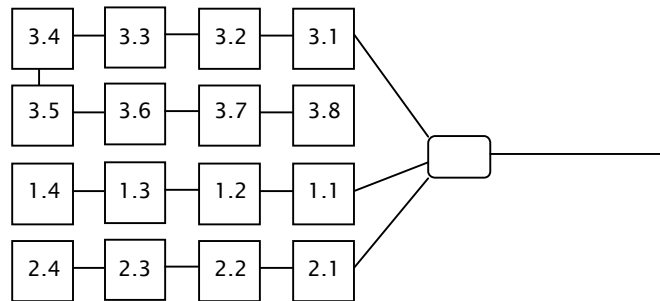
Xudp supports the following arguments for configuring the tiling arrangement:

- prefix=
  - First two octets of the IP address to send data to
  - Default is **192.168**
- port=
  - UDP port to output data to
  - Default is **306**
- tilewidth=
  - Width in pixels of an LED tile
  - Default is **16**
- tileheight=
  - Height in pixels of an LED tile

- Default is **32**
- chainstart=
  - Specifies the start of a new chain in "x\_y" format
- chainpoint=
  - Specifies a node on a chain in "x\_y" format
- noinit
  - Useful for simple demonstrations, sends data from only the first tile (0,0), to the default host address **prefix.0.254**
  - Default is disabled.
- sim=
  - Useful for simulating large network traffic or large screen sizes

### 7.3 Example Setup

Consider a 128x128 pixel display, split into a 4x4 array of tiles, each controlling 32x32 pixels. Data comes into the system from the right, and is split through a switch into 3 tile chains. Autoconfiguration may set up IP addresses as shown below (see the Autoconfiguration section 9).



To run this system, the following command should be run:

```
mplayer.exe -vf scale=128:128 -loop 0 -vo xudp:tilewidth=32:tileheight=32:  
chainstart=3_2:chainpoint=0_2:chainstart=3_3:chainpoint=0_3:  
chainstart=3_0:chainpoint=0_0:chainpoint=0_1:chainpoint=3_1 VIDEFILE.AVI
```

(Note: `-vf scale=128:128` is an optional field to software scale the input video to the correct resolution)

## 8 Data Protocol

Video and control data is transmitted from Host the (for example PC, Tiling Engine) to the scan board using a UDP based protocol. Every board boots to a default IP address of **192.168.0.254**, and the default port for communication is port **306**. See the Autoconfiguration section (9) for details of how IP addresses are assigned.

### 8.1 Packet Formats

#### 8.1.1 Header

Each packet begins with 6 bytes of header:

Bytes	Field	Default Value
0:3	Magic Word	'X' 'M' 'O' 'S'
4:5	Identifier	n/a

The identifier specifies the type of packet to follow:

Identifier	Packet Type
0x02	Data
0x03	Latch
0x04	Gamma Adjust
0x05	Intensity Adjust
0x06	Soft Intensity Adjust
0x07	Reset
0x08	AC Initiator
0x09	AC Response
0x0A	AC Finished
0x0B	AC Set IP
0x0C	Soft Intensity Adjust to Pixel
0x0D	Change Driver Type

### 8.1.2 Data

Data packets with an identifier of 0x02, include the following packet format:

Bytes	Field	Details
0:1	Tile Width	Specifies the width of the tile this packet is targeting. This field is optional, and a value of 0x0 should be used if it is to be ignored.
2:3	Tile Height	Specifies the height of the tile this packet is targeting. This field is optional, and a value of 0x0 should be used if it is to be ignored.
4:5	Reserved	Set to 0x0.
6:7	Pixel Pointer	Specifies the starting address in the pixel buffer that this packet should overwrite.
8:9	Data Length	Specifies the size in bytes of the payload. Scan boards expect 3 bytes (R,G,B) per pixel.
10:11	Dummy	Used for word alignment. Set to 0x0.
12	Data	Data field for pixel data. 24 bits per pixels expected.

### 8.1.3 Latch

Latch packets with an identifier of 0x03, include no other packet data. These packets signify the end of the frame data, and for tiles to switch from the previous frame buffer to the next.

### 8.1.4 Gamma Adjust

Gamma adjustment packets with an identifier of 0x04, include the following packet format.

Bytes	Field	Details
0	Channel	Specifies colour channel to be adjusted. 'R' for Red, 'G' for Green, 'B' for Blue and 'A' for All.
1	Dummy	Set to 0x0.
2:513	Gamma Data	Contains the gamma lookup table to store in the scan board. This is a 8bit to 16bit lookup table of 512 bytes.

### 8.1.5 Intensity Adjust

Intensity adjustment packets with an identifier of 0x05, include the following packet format:

Bytes	Field	Details
0:1	Reserved	Set to 0x0
2:3	Intensity Data	Contains the value to be written to the current gain adjustment register of the LED driver ICs.

### 8.1.6 Soft Intensity Adjust

Soft Intensity adjustment packets with an identifier of 0x06, include the following packet format:

Bytes	Field	Details
0	Channel	Specifies colour channel to be adjusted. 'R' for Red, 'G' for Green, 'B' for Blue and 'A' for All.
1	Dummy	Set to 0x0
2:3	Intensity Data	Contains the intensity adjustment value to be stored in the scanboard. Valid values are 0-255.

### 8.1.7 Soft Intensity Adjust to Pixel

Soft Intensity adjustment packets with an identifier of 0x0C, include the following packet format:

Bytes	Field	Details
0	Channel	Specifies colour channel to be adjusted. 'R' for Red, 'G' for Green, 'B' for Blue and 'A' for All.
1	X	X position of the pixel to change
2	Y	Y position of the pixel to change
3	Dummy	Set to 0x0
4:5	Intensity Data	Contains the intensity adjustment value to be stored in the scanboard. Valid values are 0-255.
6:7	Dummy	Set to 0x0

### 8.1.8 Reset

Reset packets with an identifier of 0x07, include no other packet data. These packets instruct the scan board to perform a chip reset.

### 8.1.9 Autoconfiguration

Autoconfiguration packets, with identifiers of 0x08,0x09,0x0A,0x0B, are explained in more detail in the Autoconfiguration section (9).

#### 8.1.10 Change Driver Type

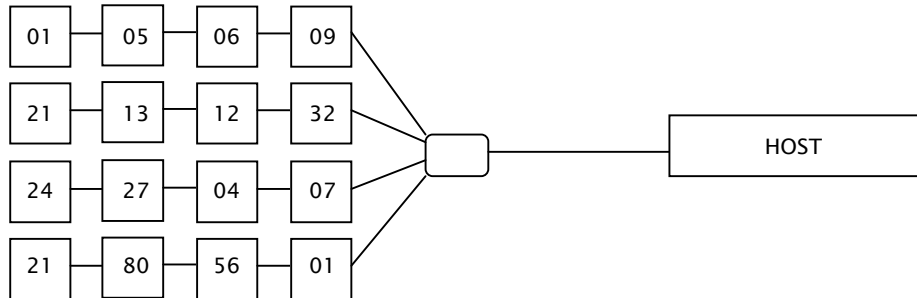
Change driver type packets, with an identifier of 0x0D, instruct the node to change its LED driver.

Bytes	Field	Details
0	Drivertype	New Driver Type: 1=MB15026 2=MB15030
1:3	Dummy	Set to 0x0

## 9 Autoconfiguration

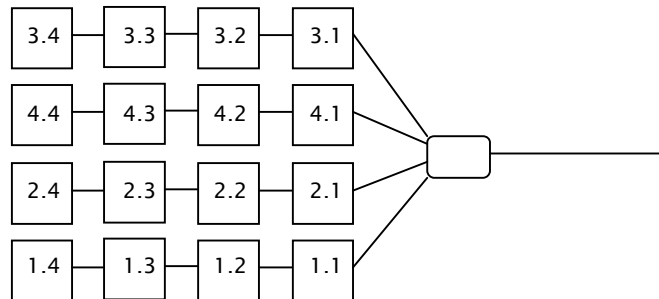
In practical systems, multiple nodes exist on the network, off multiple Ethernet chains. Automatically configuring IP addresses on this network is of benefit to system installers and users.

Consider an example where four Ethernet chains are connected to a switch, with four nodes daisy chained on each node. The diagram below shows this, and shows the last octet of the MAC address for each node. For this example, assume all other octets of the MAC address are identical.



Running an autocalibration command on the host, will allow IP addresses to be assigned along chains, and across chains, in a sensible order. This has limitations—the host and the network is unable to determine the physical order of chains attached to the switch, and can only arbitrarily order those.

Along chains, the last octet of the IP address is incremented (starting at 1 on the node nearest the source). Across chains, the lowest MAC address has its third IP address octet set to 1. The next chain (in order of MAC address), increments its third octet. The diagram below shows the third and fourth octets of the IP addresses, after an autoconfiguration packet.



## 9.1 Packet Types

The autoconfiguration system uses four packet types. All packets are broadcast on the network on UDP.

- Autoconfig Initiator

- Identifier: 0x08
  - Payload: None
  - TTL: Arbitrary
  - This packet is broadcast from the host to initiate at autoconfiguration cycle, and signals for any nodes receiving the packet, to broadcast a response.
- Autoconfig Response
    - Identifier: 0x09
    - Payload: myInitiatorTTL (byte)
    - TTL: 255
    - This packet is broadcast from any nodes receiving an initiator packet. The payload contains the Time-To-Live value from the initiator packet, and the packet's Time-To-Live must be set to maximum (255).
    - All nodes on the network will use this packet to calculate their position in the network topology.
- Autoconfig Finished
    - Identifier: 0x0A
    - Payload: none
    - TTL: Arbitrary
    - This packet is broadcast from the host a set time after the initiator packet, and signals the completion of the autoconfiguration cycle. This will begin the waterfall of IP setting packets.
- Set IP
    - Identifier: 0x0B
    - Payload: destMac (6xbyte), newIP (4xbyte)
    - TTL: Arbitrary
    - This packet is initially broadcast by the starting node, and then by the following nodes, to set the IP addresses of their neighbours.

## 9.2 Autoconfiguration Method

The core of the algorithm is each node's identification of its position based on the reception of **Autoconfig Response** packets. The node initially assumes it is the only node on a chain, and is the first node across chains.

- If the node receives a packet with a TTL of 255, it knows it has received it from a neighbour on the same chain.
  - If the *myInitiatorTTL* payload of this packet is greater than its own *myInitiatorTTL*, then the packet was received from a node before it on the chain.
  - If the *myInitiatorTTL* payload of this packet is less than its own *myInitiatorTTL*, then the packet was received from a node after it on the chain.
- If the node receives a packet with a *myInitiatorTTL* identical to its own *myInitiatorTTL*, and the node believes itself first on a chain, then it should use the MAC address of the packet to order itself.

Once this stage is complete, each node knows if it is:

- the first node on a chain or not
- the last node on a chain, and if not, what the MAC address of the next node is
- the first chain or not
- the last chain or not, and what the MAC address is of the first node on the next chain is

On receipt of an *autoconfig finished* packet, the node which is the first on a chain, and on the first chain, sets its IP address to **1.1**. This node then uses the *Set IP* packet to instruct the next node along the chain to an IP of **1.2**, and the first node on the next chain to **2.1**. This process continues down across chains, and down each chain.

## 10 Memory Requirements

### 10.1 Total Required Memory

Required RAM is shown in bytes in the following table:

Core 0	25060
Core 1	1692
Core 2	28140
Core 3	14328

### 10.2 Detailed Required Memory

Required RAM for code is shown in bytes in the following table:

Core 0	7912
Core 1	1370
Core 2	4268
Core 3	7500

Required RAM for static initialised data is shown in bytes in the following table:

Core 0	6648 (includes XMOS logo)
Core 1	298
Core 2	1168
Core 3	400

Required stack space for dynamic data is shown in bytes in the following table:

Core 0	10500
Core 1	24
Core 2	22704
Core 3	6428

### 10.3 Key Buffer Locations

The Reference Design uses the following five buffer threads:

Buffer	Default Size	Description
Receive Buffer 0	8192	Buffers packets directly off the MII layer. Used when the switch is busy transmitting or receiving packets from other directions.
Receive Buffer 1	8192	See Receive Buffer 0 above
Input Buffer	2048	Buffers locally destined packets from the switch.
Command Buffer	2048	Buffers commands separately from pixels
Pixel Buffer	4096	Buffers pixel data for two local frames.

### 10.4 Key Look-up Table Locations

The Reference Design uses the following five buffer threads:

Table	Default Size	Description
Gamma LUT 0	2304	Provides 8bit-¿16bit LUT for gamma correction for all three colour channels.

## 11 Software Structure

The software for the LED Reference Design is organised in the directory hierarchy outlined below:

**LED Reference Design**

- | **bootloader** – Directory for XCore bootloader application
  - | **bin**
    - | **XC-3** – XCore binaries
  - | **build** – Build directory
    - | **bootloader**
    - | **common**
  - | **src** – Source code
    - | **spiflash** – Code for read of flash
    - | **otp** – Code for read of OTP
- | **ledconfig** – Directory for host-side configuration applications
  - | **bin**
    - | **macos** – Macintosh OS binary directory
    - | **microsoft** – Microsoft Windows binary directory
  - | **src** – Source code
- | **ledtile** – Directory for XCore LED Tile application
  - | **bin**
    - | **XC-3** – XCore binaries
  - | **build** – Build directory
    - | **ledtile**
    - | **common**
  - | **src** – Source code
    - | **buffers**
      - | **ledbuffer** – Code for frame buffer
      - | **pktbuffer** – Code for generic buffers
    - | **ethernet**
      - | **dualmii** – Code for Ethernet MII layer
      - | **localServer** – Code for high-level Ethernet functions
      - | **switch** – Code for layer-2 switch
    - | **led**
      - | **driver** – Code driving SPI for LED drivers
      - | **processing** – Code for gamma and intensity correction
    - | **support**
      - | **misc** – Generic supporting code
      - | **retrieveMAC** – Code for retrieving the MAC address
      - | **spiflash** – Code for read and write of flash
      - | **watchdog** – Code for watchdog timer
  - | **mplayer** – Directory for host-side media playing application
    - | **bin**
      - | **macos** – Macintosh OS binary directory
      - | **microsoft** – Microsoft Windows binary directory
    - | **patch** – Patch file for XMOS UDP protocol
    - | **src** – Patched SVN source code

## 12 Related Documents

The following documents provide more information on designing with the XC-3:

- *XC-3 Hardware Manual* [1]: provides detailed information on the XC-3 hardware components and port mapping.
- *XCore XS1 Architecture Tutorial* [2]: provides an overview of the XS1 instruction set architecture.

The most up-to-date information on the XC-3, including board schematics and product datasheets, is available from:

- <http://www.xmos.com/xc3/>

## References

- [1] XMOS Ltd. XC-3 Hardware Manual. Website, 2009. <http://www.xmos.com/published/xc3hw>.
- [2] David May and Henk Muller. XCore XS1 Architecture Tutorial. Website, 2009. <http://www.xmos.com/published/xs1tut>.

XMOS Ltd is the owner or licensee of this design, code, or Information (collectively, the “Information”) and is providing it to you “AS IS” with no warranty of any kind, express or implied and shall have no liability in relation to its use. XMOS Ltd makes no representation that the Information, or any particular implementation thereof, is or will be free from any claims of infringement and again, shall have no liability in relation to any such claims.

(c) 2009 XMOS Limited - All Rights Reserved