# Application Note

**ScanLogic Corporation**
4 Preston Court
Bedford, MA 01730
http://www.scanlogic.com

**CONVENTIONS** .......... **3**
**DEFINITIONS** .......... **3**
**REFERENCES** .......... **3**
**1. SL11 CIRCUIT APPLICATION** .......... **4**
**2. CRYSTAL OSCILLATOR BUFFERS** .......... **5**
**3. PROGRAMMING INFORMATION** .......... **6**
3.1 DEVICE PROGRAMMING .......... 6
3.2- PROGRAMMING ENDPOINTS .......... 6
3.3 SL11 MEMORY MAP .......... 7
**4. APPLICATION NOTES INFORMATION** .......... **10**
4.1- SCANNER APPLICATION BLOCK DIAGRAM .......... 10
4.2- PRINTER, MODEM, AND EXTERNAL STORAGE DEVICE APPLICATION BLOCK DIAGRAM .......... 11
4.3- IMPLEMENTATION OF SL11- USB INTERFACE TO A SCANNER DESIGN EXAMPLE. .......... 12
4.4- EPP TO USB APPLICATION .......... 13
**5. SOFTWARE APPLICATION NOTES:** .......... **14**
5.1- OVERVIEW .......... 14
5.2- TALKING TO THE SL11 .......... 14
5.3- TRANSMIT DATA TO USB HOST .......... 18
5.4- SENDING A PACKET .......... 18
5.5- HANDLING THE “DONE” INTERRUPT .......... 18
5.6- HANDLING ERRORS .......... 18
5.7- RECEIVE DATA FROM USB HOST .......... 19
5.8- RECEIVING A PACKET .......... 19
5.9- DATA TOGGLE .......... 19
5.10- HANDLING THE “DONE” INTERRUPT .......... 19
5.11- HANDLING ERRORS .......... 20
5.12- USING THE “PING-PONG” BUFFERS .......... 20
5.13- DMA OPERATION .......... 21
5.14- DMA SETUP AND OPERATION .......... 22
5.15- SETTING UP A DMA TRANSFER .......... 23
5.16- A SIMPLE “EVEN” DMA TRANSFER .......... 23
5.17- AN “ODD” DMA TRANSFER .......... 24
5.18- SL11: AUTO-INCREMENT MODE: .......... 24
5.19- APPLICATION NOTE FOR SUSPEND AND RESUME .......... 25
*5.19.1- HARDWARE REQUIREMENT FOR SUSPEND AND RESUME:* .......... *25*
*5.19.2- SOFTWARE REQUIREMENT FOR SUSPEND AND RESUME:* .......... *25*

*Table 1: SL11 and Host PC address* .......... *7*
*Table 2 EndPoint 0 uses for configuration and Vendor Specific command interface* .......... *7*
*Table 3: DATA0/DATA1* .......... *7*
*Table 4: SL11 Register Control Memory Map* .......... *7*
*Table 5: EndPoint0 –EndPoint3 Registers* .......... *8*
*Table 6: Standard USB Device Request* .......... *9*
*Table 7: Descriptor Types* .......... *9*

## CONVENTIONS

| | |
|---|---|
| 1,2,3,4 | Number without annotation are decimal |
| Dh, 1Fh, 39h | Hexadecimal numbers are followed by an "h" |
| 0101b, 010101b | Binary numbers are followed by a "b" |
| *bRequest, n* | Word in *italics* indicated terms defined by USB Specification or by this Specification |
| SL11 or SL11T?? | SL11 is for 28-pin PLCC and SL11T is for 48-pin LPQFP, which is the same funtionality. When we refer SL11's term, which also indicates for SL11T. |

## DEFINITIONS

| | |
|---|---|
| **USB** | **U**niversal **S**erial **B**us |
| **SL11H** | The SL11H is a Scanlogic Host **USB** Controller, which provides multiple functions on a single chip. **Note:** This chip does not include the CPU core. |
| **SL11** | The SL11 is 28-Pin PLCC Packet, a Scanlogic Peripheral **USB** Device Controller, which provides multiple functions on a single chip. **Note:** This chip does not include the CPU core. |
| **SL11T** | The SL11T is 48 LPQFP Packet, a Scanlogic Slave **USB** Device Controller, which provides multiple functions on a single chip. **Note:** This chip does not include the CPU core. |
| **LSB** | **L**east **S**ignificant **B**it |
| **MSB** | **M**ost **S**ignificant **B**it |

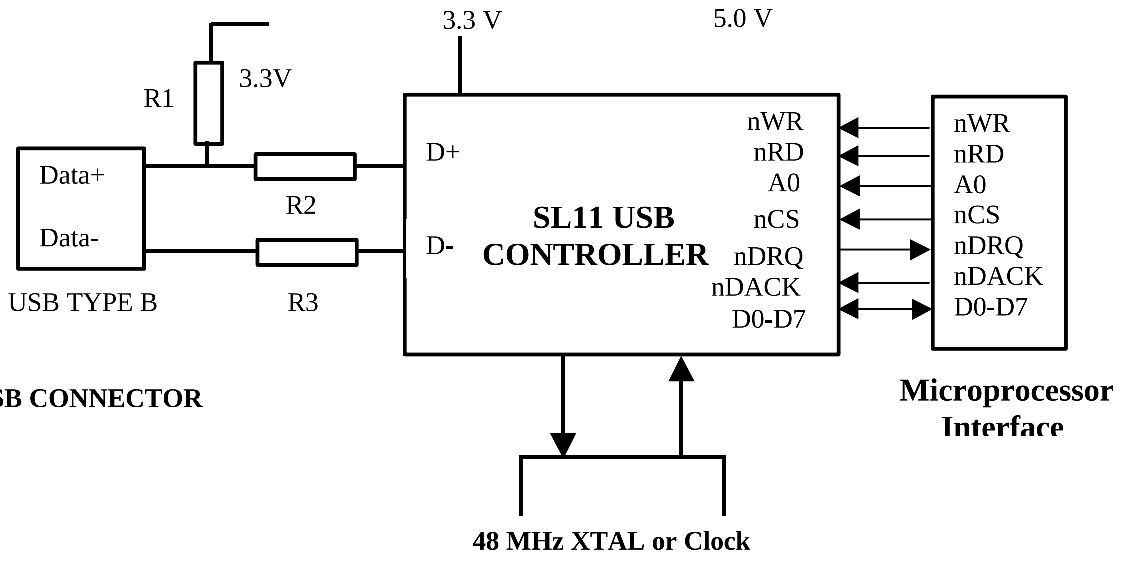
## REFERENCES

**[Ref. 1] USB Specification 1.1**
**[Ref. 2] SL11_HW: SL11 Hardware Specification**
**[Ref. 3] SL11_SWP: SL11 Software Packet**

## 1. SL11 CIRCUIT APPLICATION

A typical application for the SL11 USB Controller requires a minimum of external components. The Block Diagram shown below illustrates such an application to interface to a standard microprocessor.

**BLOCK DIAGRAM- TYPICAL SL11-MICROPROCESSOR APPLICATION**



- R1: pull-up terminator is a 1.5K resistor ±5% and is required for full speed devices (see USB Specification Ver. 1.0 and 1.1 Section 7.1.3).
- R2 and R3 are typically 24-36 Ω (see *USB Specification Ver. 1.0 and 1.1 Section 7.1.1*).
- +5.0 Volt supplied by USB is capable of source 500 mA Max.

## 2. CRYSTAL OSCILLATOR BUFFERS

There are three IO buffer types for oscillator circuitry in KZ300 series (Table 5.6.9). “XIVC” is for a real time clock application, and “XIV2” is for higher frequency from 2MHz to 20MHz range. “XIV4/XIV6” are for over 20MHz up to 50MHz. These buffers require at least two IO pads (“XIN” and “XOUT”). The dedicated VSS pad is highly recommended in general for stable oscillation (see Figure 5.6.9).

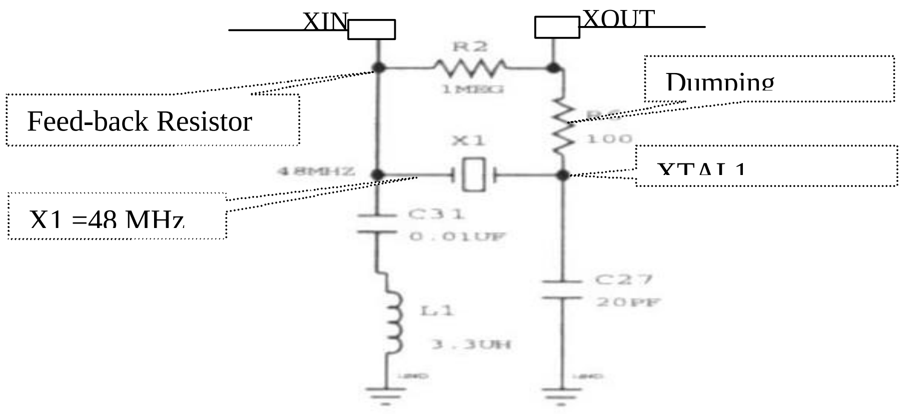
**Table 5. 6. 9:** Crystal Oscillator Buffer and Passive Device Parameters.

| Parameters | XIVC | XIV2 | XIV4/XIV6 |
|---|---|---|---|
| Rf | ≈ 8MΩ | ≈ 1MΩ | ← |
| Rd | 100kΩ-200kΩ | 0-200Ω | ← |
| Cl | 5-30 pF | 5-30 pF | ← |
| CO | 5-30 pF | 5-30 pF | ← |
| C1 | 5-30 pF | 5-30 pF | 0.01 µF |
| L | 5-30 pF | 5-30 pF | 2.2-3.3 µH |

The “XIN” and the “XOUT” pins must be assigned next to each other and “XIN” must be assigned to the lower pad location than “XOUT.” Also, placing dedicated VSS pin next to either “XIN” or “XOUT” is highly recommended.

To achieve a stable operation, the oscillator must be kept far enough from noise source. High frequency switching output buffers or high driving output buffers should not be assigned near the oscillator.

**Fig. 5. 6. 9 1: XIV4/XIV6** (20-50MHz)



# 3. PROGRAMMING INFORMATION

## 3.1 DEVICE PROGRAMMING

1.

| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | A0 |
|---|---|---|---|---|---|---|---|---|
| REG/MEMORY ADDRESS | | | | | | | | '0' |

WRITE
ADDRESS

2.

| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | A0 |
|---|---|---|---|---|---|---|---|---|
| REG/MEMORY DATA | | | | | | | | '1' |

READ/WRITE
REGISTER
OR
MEMORY

**SL11 Register/Memory I/O Operations.**
**1. Write Address to I/O location with A0 = '0'.**
**2. Read/Write data from/to location. A0 = '1'.**

The SL11 provides access to programming and data registers through an 8 bit data port using the appropriate control signals. The device may be utilized with processors that support multiplexed address and data buses or with processors that support separate address and data buses. When used with processors that require multiplexed address and data bus support, the device utilizes the **ALE** control input, along with the Read and Write inputs to access internal registers or memory .

In non multiplexed interface applications, shown in diagram above, the internal register address is input to the device in a normal I/O or memory mapped I/O write operation with the A0 address select input driven low ('0'). This operation results in the address being latched internally so that a following Read or Write operation with A0 driven high ('1') will result in a data transfer.

## 3.2- PROGRAMMING ENDPOINTS

The SL11 supports Endpoints, 0 - 3. Endpoint 0 is the default pipe, and supports control transfers. Endpoints 1 - 3 can support bulk, interrupt, or isochronous transfers. Each endpoint has two sets of control, the 'a' and 'b' set. This allows overlapped operation, where one set of parameters is being set up, while the other is transferring. Upon completion of a transfer to an endpoint, the 'next data set' bit indicates whether set 'a' or set 'b' will be in effect next. The 'armed' bit of the next data set will indicate whether the SL11 is ready for the next transfer without interruption.

Each endpoint has an associated set of registers, described in previous section, which must be programmed in order initiate or respond to transactions on the USB. Endpoint EP0 transactions are initiated by the host during setup and configuration. Typically the host will request information from the device during setup to determine the device's characteristics, and will also assign a USB ID to the device. The complete definition of control messages and transactions are defined in Chapter 9 of the USB Specification 1.1.

Once configuration has been completed, the device can be programmed to send or receive data using EndPoint1-EndPoint3.

## 3.3 SL11 MEMORY MAP

**Table 1: SL11 and Host PC address**

| Register Name | Address | Register Function |
|---|---|---|
| | | |
| SL11_ADDR | 0x390 | SL11 Address |
| PC_ADDR | 390h | Host PC Address |

**Table 2 EndPoint 0 uses for configuration and Vendor Specific command interface**

| Register Name | Address | Register Function |
|---|---|---|
| | | |
| EP0Buf | 0x40 | Endpoint 0 Buffer where SL11 memory start |
| EP0Len | 0x40 | Length of config buffer EP0Buf |
| EP1Buf | 0x60 | EndPoint 1 Buffer |
| EP1Len | 0x40 | Length of config buffer EP1Buf |

**Table 3: DATA0/DATA1**

SL11 memory from 80h-ffh use as ping-pong buffer for EndPoint 1- EndPoint 3. These buffers are shared for EndPoint 1- EndPoint 3. For DMA, EndPoint3 will be used

| Register Name | Address | Register Function |
|---|---|---|
| | | |
| uBufA | 0x80 | Buffer A address for DATA0 |
| uBufB | 0xc0 | Buffer B address for DATA1 |
| uXferLen | 0x40 | Xfer Length |
| sMemSize | 0xc0 | Total SL11 memory size |

**Table 4: SL11 Register Control Memory Map**

| Register Name | Address | Register Function |
|---|---|---|
| | | |
| CtrlReg | 0x05 | Control Register |
| IntStatus | 0x0d | Interrupt Status |
| USBAdd | 0x07 | USB Address |
| IntEna | 0x06 | Interrupt Enable |
| DATASet | 0x0e | DATA Set |
| sDMACntLow | 0x35 | DMA Count Low |
| sDMACntHigh | 0x36 | DMA Count High |
| InMask | 0x57 | Rest\|DMA\|EP0\|EP1\|EP2 for IntEna |
| HostMask | 0x47 | Host request command for IntStatus |
| ReadMask | 0xd7 | Read mask interrupt for IntStatus |

**Table 5: EndPoint0 –EndPoint3 Registers**

| Register Name | Address | Register Function |
|---|---|---|
| **EndPoint 0** | | |
| EP0Control | 0x00 | EndPoint 0 Control Register |
| EP0Address | 0x01 | EndPoint 0 Address Register |
| EP0XferLen | 0x02 | EndPoint 0 xfer length Register |
| EP0Status | 0x03 | EndPoint 0 Status Register |
| EP0Counter | 0x04 | EndPoint 0 Counter Register |
| **EndPoint 1 A and B** | | |
| EP1AControl | 0x10 | EndPoint 1A Control Register |
| EP1AAddress | 0x11 | EndPoint 1A Address Register |
| EP1AXferLen | 0x12 | EndPoint 1A xfer length Register |
| EP1ACounter | 0x14 | EndPoint 1A Counter Register |
| | | |
| EP1BControl | 0x18 | EndPoint 1B Control Register |
| EP1BAddress | 0x19 | EndPoint 1B Address Register |
| EP1BXferLen | 0x1a | EndPoint 1B xfer length Register |
| EP1BCounter | 0x1c | EndPoint 1B Counter Register |
| **EndPoint 2 A and B** | | |
| EP2AControl | 0x20 | EndPoint 2A Control Register |
| EP2AAddress | 0x21 | EndPoint 2A Address Register |
| EP2AXferLen | 0x22 | EndPoint 2A xfer length Register |
| EP2ACounter | 0x24 | EndPoint 2A Counter Register |
| | | |
| EP2BControl | 0x28 | EndPoint 2B Control Register |
| EP2BAddress | 0x29 | EndPoint 2B Address Register |
| EP2BXferLen | 0x2a | EndPoint 2B xfer length Register |
| EP2BCounter | 0x2c | EndPoint 2B Counter Register |
| **EndPoint 3 A and B** | | |
| EP3AControl | 0x30 | EndPoint 3A Control Register |
| EP3AAddress | 0x31 | EndPoint 3A Address Register |
| EP3AXferLen | 0x32 | EndPoint 3A xfer length Register |
| EP3ACounter | 0x34 | EndPoint 3A Counter Register |
| | | |
| EP3BControl | 0x38 | EndPoint 3B Control Register |
| EP3BAddress | 0x39 | EndPoint 3B Address Register |
| EP3BXferLen | 0x3a | EndPoint 3B xfer length Register |
| EP3BCounter | 0x3c | EndPoint 3B Counter Register |

**Table 6: Standard USB Device Request**

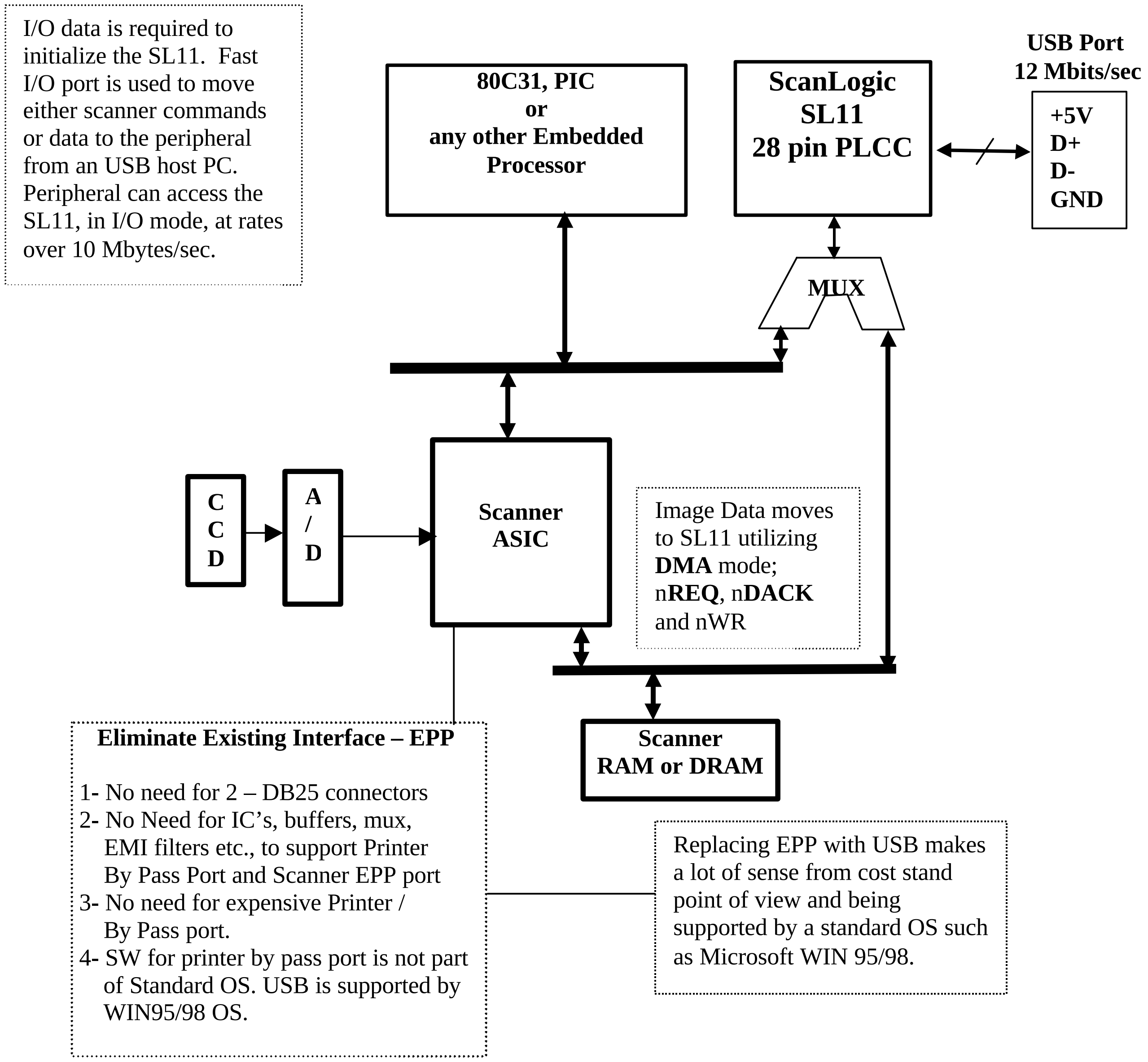
| Device Request | Support | Address | Function |
|---|---|---|---|
| | | | |
| GET_STATUS | Yes | 0x00 | Return device, interface, or EndPoint Status |
| CLEAR_FEATURE | Yes | 0x01 | Allows the DEVICE_REMOTE and ENDPOINT_STALL features to be cleared |
| SET_FEATURE | Yes | 0x03 | Allows the DEVICE_REMOTE_WAKEUP and ENDPOINT_STALL features to be set |
| SET_ADDRESS | Yes | 0x05 | Set the device address for devices |
| GET_DESCRIPTOR | Yes | 0x06 | Uses to retrieve Device, configuration, string descriptors from the device |
| SET_DESCRIPTOR | Yes | 0x07 | |
| GET_CONFIG | Yes | 0x08 | Returns the current device configuration. |
| SET_CONFIG | Yes | 0x09 | Sets the device configuration. |
| GET_INTERFACE | Yes | 0x0a | Returns the selected alternate setting for the specified interface. |

**Table 7: Descriptor Types**

| Descriptor Types | Address |
|---|---|
| | |
| DEVICE | 0x01 |
| CONFIGURATION | 0x02 |
| STRING | 0x03 |
| INTERFACE | 0x04 |
| ENDPOINT | 0x05 |

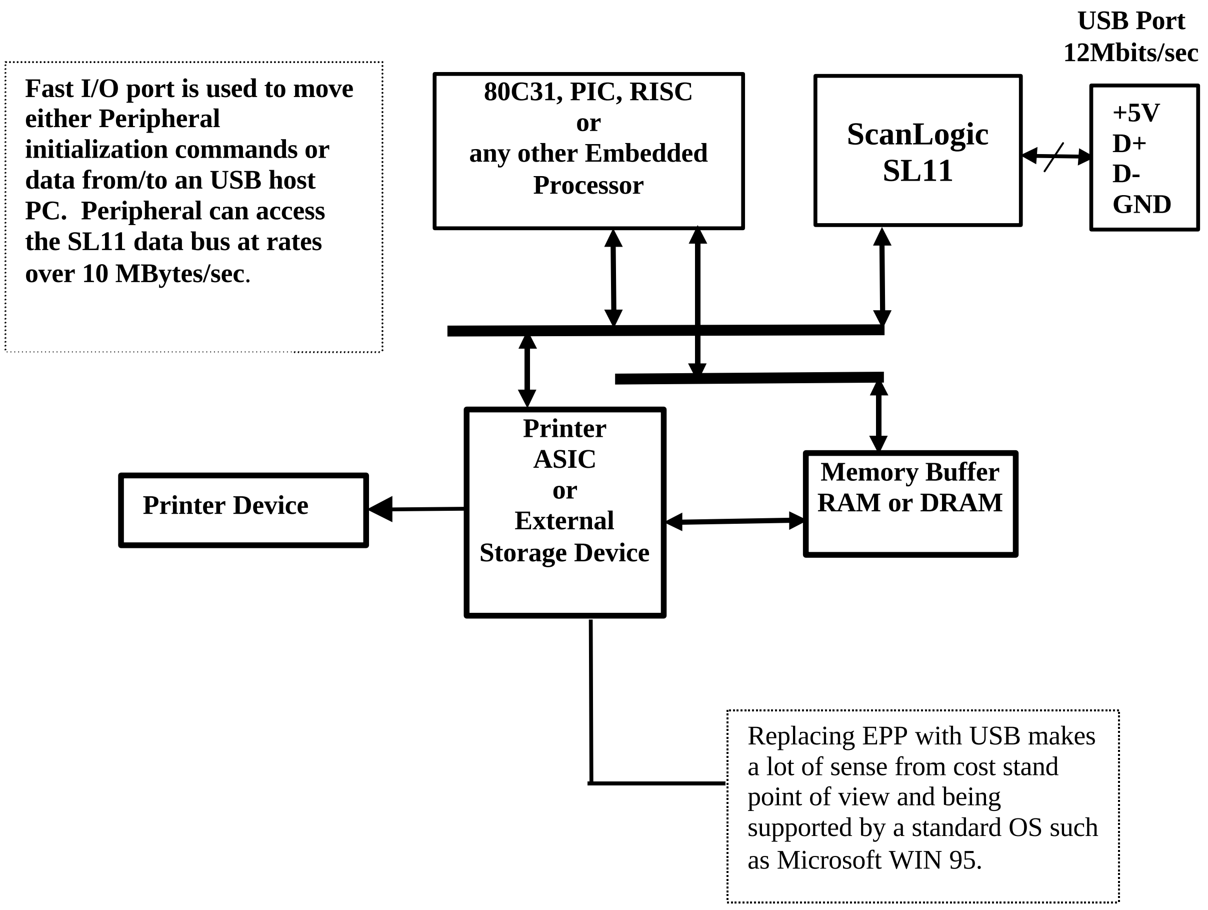
# 4. APPLICATION NOTES INFORMATION

## 4.1- SCANNER APPLICATION BLOCK DIAGRAM


I/O data is required to initialize the SL11. Fast I/O port is used to move either scanner commands or data to the peripheral from an USB host PC. Peripheral can access the SL11, in I/O mode, at rates over 10 Mbytes/sec.
80C31, PIC or any other Embedded Processor
ScanLogic SL11 28 pin PLCC
USB Port 12 Mbits/sec
+5V
D+
D-
GND
MUX
C C D
A / D
Scanner ASIC
Image Data moves to SL11 utilizing DMA mode; nREQ, nDACK and nWR
Scanner RAM or DRAM
Eliminate Existing Interface – EPP
1- No need for 2 – DB25 connectors
2- No Need for IC's, buffers, mux, EMI filters etc., to support Printer By Pass Port and Scanner EPP port
3- No need for expensive Printer / By Pass port.
4- SW for printer by pass port is not part of Standard OS. USB is supported by WIN95/98 OS.
Replacing EPP with USB makes a lot of sense from cost stand point of view and being supported by a standard OS such as Microsoft WIN 95/98.

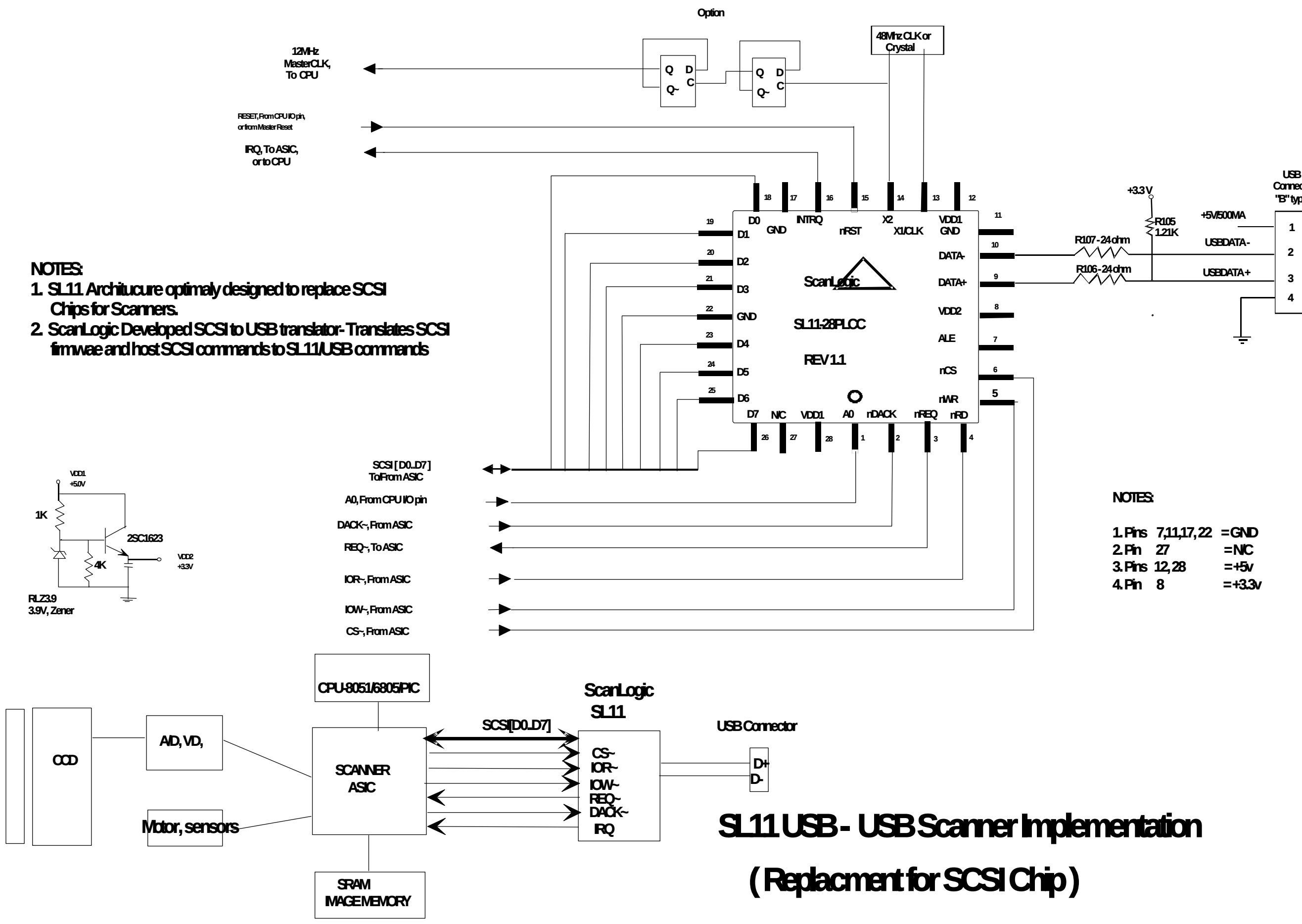
**ScanLogic Corporation – SL11 USB Interface Chip**
**Scanner Application Block Diagram**

## 4.2- PRINTER, MODEM, AND EXTERNAL STORAGE DEVICE APPLICATION BLOCK DIAGRAM


USB Port
12Mbits/sec
Fast I/O port is used to move either Peripheral initialization commands or data from/to an USB host PC. Peripheral can access the SL11 data bus at rates over 10 MBytes/sec.
80C31, PIC, RISC or any other Embedded Processor
ScanLogic SL11
+5V
D+
D-
GND
Printer ASIC or External Storage Device
Printer Device
Memory Buffer RAM or DRAM
Replacing EPP with USB makes a lot of sense from cost stand point of view and being supported by a standard OS such as Microsoft WIN 95.

**ScanLogic Corporation – SL11 USB Interface Chip**
**Printer, Modem, Ext. Storage Device Application**
**Block Diagram**

## 4.3- IMPLEMENTATION OF SL11- USB INTERFACE TO A SCANNER DESIGN EXAMPLE.



NOTES:
1. SL11 Architucure optimaly designed to replace SCSI Chips for Scanners.
2. ScanLogic Developed SCSI to USB translator- Translates SCSI firmwae and host SCSI commands to SL11/USB commands

NOTES:

| | | |
|---|---|---|
| 1. Pins | 7,11,17,22 | = GND |
| 2. Pin | 27 | = N/C |
| 3. Pins | 12, 28 | = +5v |
| 4. Pin | 8 | = +3.3v |

**SL11 USB - USB Scanner Implementation**

**( Replacment for SCSI Chip )**

## 4.4- EPP TO USB APPLICATION

This is an example of how to design an interface between EPP to the SL11/USB (see **Figure below**). The GAL16V8 is used as an optional translator between EPP timing to SL11 **DMA** timing requirements. For the sake of HW debugging, it is suggested that designer should use the GAL16V8 for initial HW development.

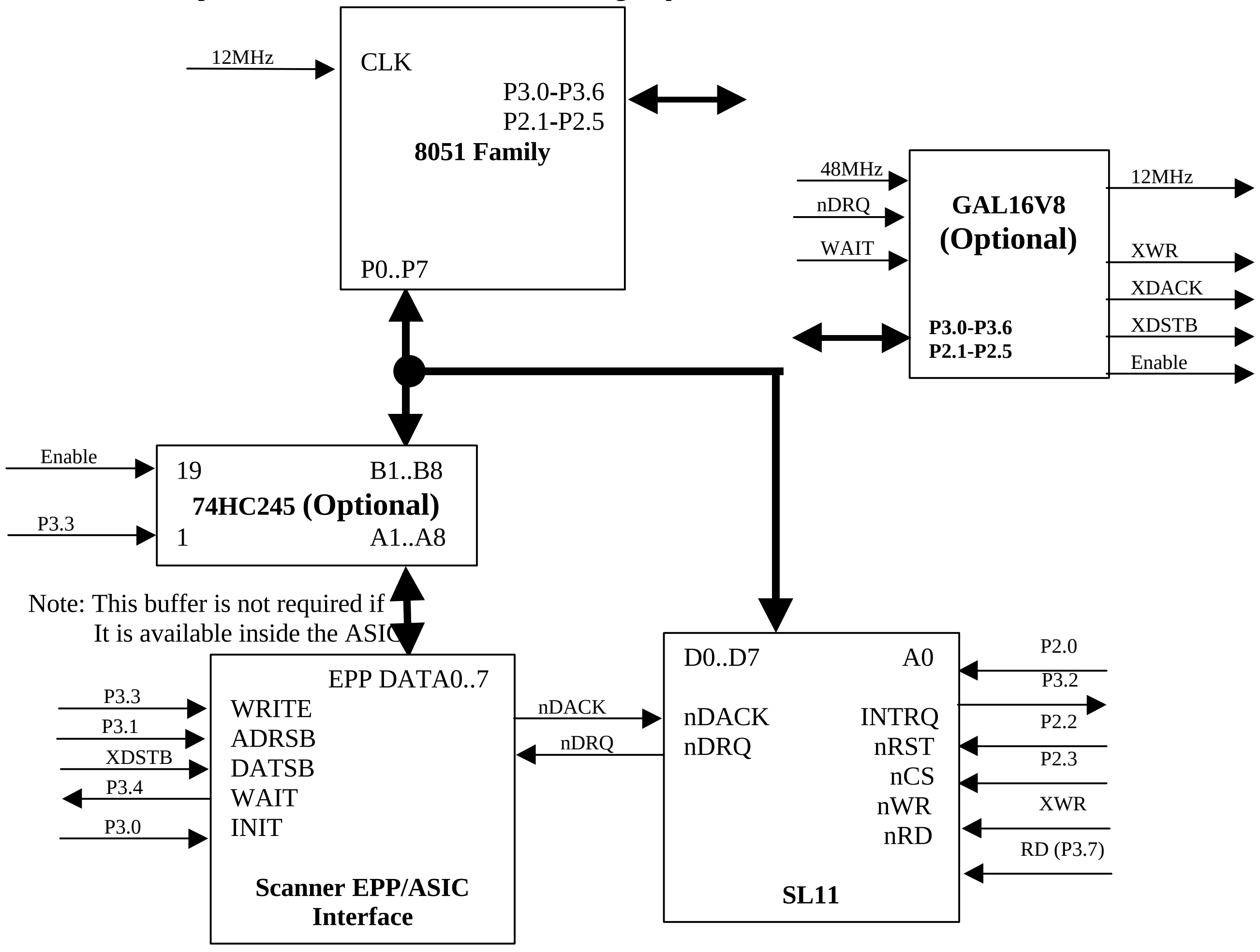
The 74HC245 can be removed, if the Firmware correctly controls the common data bus.

The GAL16V8 can be replaced by any conventional available gates, which may reduce overall cost. User can choose any Micro controller. In this example, the 8051 family is used to demonstrate the flexibility of the SL11 chip.

Equation example in the GAL16V8:

```
    Enable = ! P2.3;   (nCS)   ; which mean whenever CPU accessing the SL11 the Output of
                               ; 74HC245 should be tri-state.
                               ; When CPU accessing the Scanner EPP, the 74HC245 will be
                               ; changed from B->A.
```

See the SL11 specification for detail of **DMA** timing requirement.



# 5. SOFTWARE APPLICATION NOTES:

## 5.1- OVERVIEW

The purpose of this document is to provide a basic guide to programming the SL11 USB Controller. The next section will illustrate how a Micro-controller might communicate with the SL11. In successive sections, examples will be shown for three possible operations to be performed with the SL11:

- Transmit Data
- Receive Data
- DMA Transfer

## 5.2- TALKING TO THE SL11

The exact hardware interface used to communicate with the SL11 is beyond the scope of this document, and can vary depending on what device the SL11 is connected to. This section describes the process only in general terms.

The process to initialize function of the SL11 Chip is setup USB addresses, EndPoint0, EndPoint1, Endpoint2, enable USB, and clear all the interrupt status flags.

**Example in C-language:**

```
void SL11Init()
{
    dConf = dAddress = 0;
    SL11Write(USBAdd,0x00);           //usb address
    SL11Write(EP0Address,EP0Buf);     //data addr
    // EP0Receive
    SL11Write(EP0XferLen,EP0Len);
    SL11Write(EP0Control,0x03);       //set Receive data from host

    SL11Write(EP1AAddress,EP1Buf);    //data addr
    SL11Write(EP1AXferLen,EP1Len);
    SL11Write(EP1AControl,0x3);

    SL11Write(EP2AAddress,uBufA);     //data addr
    SL11Write(EP2BAddress,uBufB);
    SL11Write(EP2AControl,0x3);
    SL11Write(EP2AXferLen,uXferLen);  //data addr

    SL11Write(CtrlReg,1);             //enable USB
    SL11Write(IntStatus,0xff);        // Clear the interrupt all flags
```

**Example in Scenix Assembler-language:**

```
SL11Init                      ; SL11 Init function
 bank  bank10                 ;USB standard commond packets
 clr   dAddress               ;clear aAddress
 clr   ConfigReg
 mov   b,#USBAdd              ;a,b are local variable
 clr   a                      ;set address = 0
 call  @SL11Write
 mov   b,#EP0Counter
 call  @SL11Write             ;setup EndPoint0
 mov   b,#EP0Address
 mov   a,#EP0Buf              ;Setup EP0 Buffer
 call  @SL11Write
 call  @EP0Receive
 mov   b,#EP1AAddress
 mov   a,#uBufA               ;DATA0 from 0x80
 call  @SL11Write
 mov   b,#EP1BAddress
 mov   a,#uBufB               ;DATA1 from 0xc0
 call  @SL11Write
 mov   b,#CtrlReg
 mov   a,#1h                  ;enable USB
 call  @SL11Write
 mov   b,#IntEna
 mov   a,#IntMask             ;set interupt mask
 call  @SL11Write
 mov   b,#IntStatus
 mov   a,#0ffh                ;Clear the interrupt all flags
 jmp   @SL11Write             ;call SL11Write and ret

; Setup to receive EP0Len from host
EP0Receive
 mov   b,#EP0XferLen
 mov   a,#EP0Len
 call  SL11Write
 mov   b,#EP0Control        ; set Receive data from host
 mov   a,#03h
 jmp   SL11Write
```

Writing a byte to the SL11 involves two writes cycles: First, the application must write a register address into the SL11's Address Pointer Register. Second, the actual data is written to the chip in a second write. Reading a byte from the SL11 involves a write cycle followed by a read cycle: First, as in a data write, the application writes a register address into the SL11's Address Pointer Register; Second, the data is read from the chip in a read cycle.

In memory-mapped or I/O-mapped applications, the A0 pin on the SL11 is used to select the Address Register or the Data register. In Micro-controllers that multiplex the address and data buses, the **ALE** pin is used to de-multiplex the address and data buses. For a detailed description of the hardware interface, please refer to the SL11 USB Controller Specification.

In an I/O mapped application, a function to write a byte and write a buffer to an SL11 register might read like this:

```
 // Byte Write to SL11
 // Input:  a = address of SL11 register or memory, d = data

void SL11Write(BYTE a, BYTE d)
{
#ifdef USEC
   outportb(SL11_ADDR,a);            // SL11_ADDR = 0x390
   outportb(SL11_ADDR+1,d);
#else
   _asm {
       mov edx,PC_ADDR               // PC_ADDR = 390h
       mov al,a
       out dx,al
       inc dx
       mov al,d
       out dx,al
  }
#endif
}

// Buffer Write  to SL11
// Input:  a = address of SL11 register or memory
//         c = byte count, s = char buffer

void SL11BufWrite(BYTE a, BYTE *s, short c)
{
   if(c<=0) return;
#ifdef USEC
   outportb(SL11_ADDR,a);
   while (c--) outportb(SL11_ADDR+1,*s++);
#else
   _asm {
   mov   edx,PC_ADDR
   mov   al,a
   out   dx,al
   inc   dx
   cld
   mov   esi,s
   movzx ecx,c
   rep   outsb
   }
#endif
}
```

And a function to read an SL11 register might look like this:

```
// Byte Read from SL11
// Input:  a = address of SL11 register or memory
// Output: return data

BYTE SL11Read(BYTE a)
{
#ifdef USEC
   outportb(SL11_ADDR,a);            // SL11_ADDR = 0x390
   return (inportb(SL11_ADDR+1));
#else
   _asm {
   mov edx,PC_ADDR                   // PC_ADDR = 390h
   mov al,a
   out dx,al
   inc dx
   in  al,dx
   }
#endif
}

// Buffer Read from SL11
// Input:  a = address of SL11 register or memory
//         c = byte count, s = char buffer

void SL11BufRead(BYTE a, BYTE *s, short c)
{
   if(c<=0) return;
#ifdef USEC
   outportb(SL11_ADDR,a);
   while (c--) *s++ = (BYTE)inportb(SL11_ADDR+1);
#else
   _asm {
   mov   edx,PC_ADDR
   mov   al,a
   out   dx,al
   inc   dx
   cld
   mov   edi,s
   movzx ecx,c
   rep insb
   }
#endif
}
```

## 5.3- TRANSMIT DATA TO USB HOST

In this example, we want to send a packet of 8 bytes from Endpoint 1-a to the host.

## 5.4- SENDING A PACKET

Assuming that the data we want to send is already loaded into the SL11's buffer, starting at address 68 hex, the sequence would be:

```
base = 0x10               //  hex 10 is base register for endpoint 1-a
SL11write (base+1,0x68)   //  load base address register with data addr
SL11write (base+2,8)      //  load data length into length register
x = SL11read(0x06)        //  get contents of interrupt enable register
SL11write (0x06,x | 2)    //  OPTIONAL: enable endpoint 1 done interrupt
SL11write (base,7)        //  sets the following bits:
                          //  bit 0 = 1: Arm this endpoint
                          //  bit 1 = 1: Enable this endpoint
                          //  bit 2 = 1: Direction is Transmit
                          //  bit 3 = 0:  Next data set will also be 1-a
                          //  bit 6 = 0: Sequence is DATA0
```

For subsequent transmissions, bit 6 should get toggled between “1” and “0” to ensure that the USB Host sees correct data sequencing.

## 5.5- HANDLING THE “DONE” INTERRUPT

If the interrupt is enabled, the application must include an interrupt handler that would process the interrupt, possibly in this manner:

```
x = SL11read (0x0D)           // read the interrupt status register
if (x & 2)                    // interrupt caused by Endpoint 1 done
{
      status = SL11read (base + 3)  // read the packet status
      if (status & 1)               // Transmission acknowledged
            // everything should be OK
            // toggle the sequence bit for the next send
      else
            // examine other bits to determine error condition
}
```

## 5.6- HANDLING ERRORS

The Packet Status register (base+3) defines several bits for various error conditions. The error condition most likely to be encountered on a packet send is “Timeout,” bit 2 of the Packet Status Register. A Timeout would indicate that the USB host encountered an error, and did not acknowledge the sent packet. In this case, the most appropriate action would be just to re-arm the endpoint, so that the packet will be retransmitted. To continue the above code fragment:

```
if ( status & 4 ){
      x = SL11read ( base ) // get the contents of the control register
      SL11write ( base, x | 1 )// reset the ‘armed’ bit
}
```

## 5.7- RECEIVE DATA FROM USB HOST

In this example, we expect a packet of up to 64 bytes (40 hex) to be sent to endpoint 0-a from the USB host.

```
void EP0Receive(){
    SL11Write(EP0XferLen,EP0Len);
    SL11Write(EP0Control,0x03);
}
```

## 5.8- RECEIVING A PACKET

Assuming that we want the data to be placed in the SL11's buffer starting at address 60 hex, the sequence would be:

```
base = 0                 // this is the base address for endpoint 0-a
SL11write ( base + 1, 0x60 )  // buffer address into which to place data
SL11write ( base + 2, 0x40 )  // maximum length of data to be received
x = SL11read ( 0x06 )      // get contents of interrupt enable register
SL11write ( 0x06, x | 1 )  // OPTIONAL: enable Endpoint 0 interrupt
SL11write ( base, 3 )      // sets the following bits:
                           // bit 0 = 1: Arm this endpoint
                           // bit 1 = 1: Enable this endpoint
                           // bit 2 = Direction: low = receive from host
                           // bit 3 = 0: Next data set is also 0-a
                           // bit 6 = 0:  we expect DATA0 sequence
```

For alternate packets, bit 6 should be toggled between “1” and “0” to ensure correct data sequencing.

## 5.9- DATA TOGGLE

If you use "A" Endpoint for DATA0 and DATA1 packet then wait for the transfer done before you toggle the bit. For example: SL11Write (EP0Control, 3), which is arm to send DATA0 on endpoint 0
**Note:** the B endpoint Control should no be enabled.

You will receive the interrupt and the ARM bit on the EP0Control will be cleared. You should check error, timeout etc. on Packet Status.

```
SL11Write (EP0Control, 0x43)          ; arm to send DATA1 on endpoint 0
```

If you use the A endpoint for DATA0 and the B endpoint for DATA1 packet then you also need to wait for the transfer done before you set ARM on the other endpoint See "sl11.c" example source code.

## 5.10- HANDLING THE “DONE” INTERRUPT

If the interrupt is enabled, the application must include an interrupt handler to process the interrupt, possibly in this manner:

```
x = SL11read (0x0D)      // read the interrupt status register
if ( x & 1 )             // yes, interrupt caused by Endpoint 0 done
{
      status = SL11read ( base + 3 )// read the packet status
      if ( status & 1 )            // Transmission acknowledge
            // everything OK
            // flip the sequence bit in control register for next
      else
```

```
                // examine other bits to determine error condition
    }
```

For packets received from the host, the application can also poll the **T**ransfer **C**ount **R**egister at (base + 4) to determine the number of bytes actually received. Because the Transfer Count register contains the number of bytes left in the buffer at the end of a transfer, the calculation would look like this.

```
    actual_bytes_received = 0x40 – SL11read(base + 4)
```

If the amount of data received is not what the application expected, the application will have to take appropriate action.

## 5.11- HANDLING ERRORS

Again, we will want to examine the Packet Status Register (base+3) to determine the cause of the error. For a received packet, the most likely error is a CRC error, which would cause bit 2 to be set.
To continue the above code fragment:

```
    if ( status & 2 )
    {
          x = SL11read (base)         // get the control register contents
          SL11write ( base, x | 1 )  // and rearm the endpoint
    }
```

The host will re-send the same packet if it does not receive an Acknowledge within a certain period of time.

## 5.12- USING THE "PING-PONG" BUFFERS

The SL11 provides two sets of registers for each endpoint. They are referred to in this document and in the SL11 USB Controller Specification as endpoints *n*-a and *n*-b. This allows overlapping operation, where one set of registers can be set up while data is being transferred under control of the other set.

The control register for each endpoint includes a 'next data set' bit, which specifies which endpoint the next data set will be sent to, as well as a 'sequence' bit, which specifies whether this endpoint should receive DATA0 or DATA1 packets. If you are going to use the ping-pong buffers, the 'next-data-set' bit in each endpoint's control register should be set to point to the other endpoint.

Refer to the SL11 USB Controller specification for a precise definition of these bits.

Note that use of the ping-pong buffers is optional. The decision to use this feature depends on the precise nature of the application, and the performance of the Micro-controller in use.

## 5.13- DMA OPERATION

The SL11 will support **DMA** transfers on the 8 bits data bus. The two signals, n**DRQ** and n**DACK**, are used along with the n**WR** signal to move data into the SL11 from the microprocessor data bus.

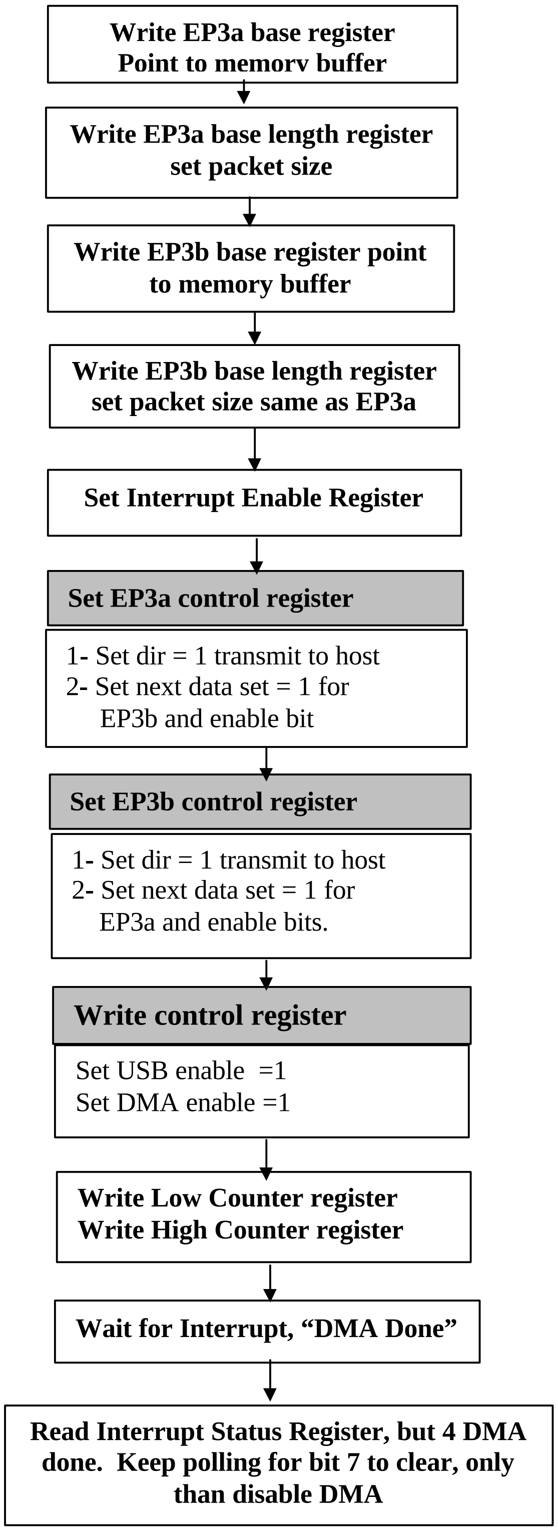
**Note:** **DMA** read transfers from the SL11 to the microprocessor bus are currently not supported.

Internally, **DMA** utilizes Endpoint 3 to move data out on the USB. Both data sets, 3a and 3b, must be set up, and must have the same length specified. Both data sets must be armed and enabled, with the 'next data set' bit pointing to the opposite data set. **DMA** total count must be loaded in the **DMA** total count registers.

For **DMA** writes, a **dreq** will be issued, and data will be stored in the first data set until full. This data set will now be ready for USB transfer. Next, data will be stored in the second data set until full. If the transfer of the first data set on USB is not complete, **DMA** operation will be suspended until it is. **DMA** will continue, without program I/O intervention required, until **DMA** total count has been reached. Then the **DMA** done bit will be set in the status register, and, an interrupt will occur if enabled

## 5.14- DMA SETUP AND OPERATION

The following flowchart represents a typical **DMA** setup and sequence of SL11's operation:


Write EP3a base register
Point to memorv buffer
Write EP3a base length register
set packet size
Write EP3b base register point
to memory buffer
Write EP3b base length register
set packet size same as EP3a
Set Interrupt Enable Register
Set EP3a control register
1- Set dir = 1 transmit to host
2- Set next data set = 1 for
EP3b and enable bit
Set EP3b control register
1- Set dir = 1 transmit to host
2- Set next data set = 1 for
EP3a and enable bits.
Write control register
Set USB enable =1
Set DMA enable =1
Write Low Counter register
Write High Counter register
Wait for Interrupt, "DMA Done"
Read Interrupt Status Register, but 4 DMA
done. Keep polling for bit 7 to clear, only
than disable DMA

## 5.15- SETTING UP A DMA TRANSFER

In this example, we want to set up a **DMA** transfer from application memory to the SL11, with the data to then be sent to the host. In the SL11, endpoint 3 is used for **DMA** operations. This example also uses the SL11's ping-pong buffers, which were ignored in the first two examples for the sake of simplicity.

## 5.16- A SIMPLE "EVEN" DMA TRANSFER

In this example, endpoint 3-a is used as DATA0 and endpoint 3-b is used as DATA1. For the sake of simplicity, we are assuming that the amount of data to be sent is evenly divisible by the data size set for the SL11's endpoint length registers. Assuming that we want to send 256 (100 hex) bytes, we might set the data length register for endpoints 3-a and 3-b to 64 (40 hex) bytes each.

```
base_3a = 0x30                // base address for EP 3-a
base_3b = 0x38                // base address for EP 3-b
x = SL11read ( 5 )            // read control register
SL11write ( 5, x | 2 )        // make sure DMA enable bit is set
SL11write ( base_3a+1, 0x80 ) // data address for EP 3-a
SL11write ( base_3b+1, 0xC0 ) // data address for EP 3-b (EP 3-a + 40h)
SL11write ( base_3a+2, 0x40 ) // data size for EP 3-a  (64 bytes)
SL11write ( base_3b+2, 0x40 ) // data size for EP 3-b  (64 bytes)
SL11write ( base_3a,   0x0E ) // EP 3-a control register:
                              // bit 0 = 1: Arm this endpoint
                              // bit 1 = 1: Enable this endpoint
                              // bit 2 = 1: Direction is transmit
                              // bit 3 = 1: Next Data Set will be 'B'
                              // bit 6 = 0.  Set Sequence bit for DATA0
SL11write ( base_3b, 0x46 )   // EP 3-b control register
                              // bit 1 = 1: Enable ths endpoint
                              // bit 2 = 1: Direction: is transmit
                              // bit 3 = 0: Next Data set will be 'A'
                              // bit 6 = 1: Set Sequence bit for DATA1
Setup_DMA                 // application does its DMA setup routine here
x = SL11read ( 0x06 )     // get contents of interrupt enable register
SL11write ( 0x06, x | 0x10 )  // enable DMA Done interrupt
SL11write ( 0x35, 0 )         // Low Byte of DMA Total Count
SL11write ( 0x36, 1 )         // High Bye of DMA Total Count
                              // total number of bytes = 256 (100h)
```

Writing to the **DMA** total count high-byte register initiates the SL11's **DMA** transfer process. The SL11 will automatically initiate **DMA** cycles to fill each buffer as required, until the Total Count number of bytes has been transferred.

The interrupt handler to process the **DMA** done interrupt might look like this:

```
x = SL11read (0x0D)           // read interrupt status register
if ( x & 0x10 )               // it's the DMA done interrupt
      do whatever is next
else
      handle other interrupts as needed
```

## 5.17- AN “ODD” DMA TRANSFER

As mentioned in the SL11 USB Controller Specification, the **DMA** mechanism will only send a whole number multiple of the packet size set in the SL11’s Endpoint Base Length register. If you need to transfer an “odd” number of bytes, the remainder has to be sent separately. All of the data will be placed in the SL11’s buffers by **DMA**, but the length of the “remainder” bytes must be set under program control.

Suppose, for example, that you need to transfer 267 (10B hex) bytes to the USB. Instead of

```
      SL11write ( 0x35, 0 )    // Low Byte of DMA Total Count
      SL11write ( 0x36, 1 )    // High Bye of DMA Total Count
```

You would need to write:

```
      SL11write ( 0x35, 0x0B ) // Low Byte of DMA Total Count
      SL11write ( 0x36, 1 )    // High Bye of DMA Total Count
```

The first 256 bytes would be sent exactly as described in section 0. However, when the **DMA** done interrupt occurs, there are still 11 bytes remaining to be sent.

To handle this case, the interrupt handler would have to read like this.

```
      if ( remainder > 0 )                 // remainder = number of bytes left
      {
             do
             {
                 x = SL11read (0x0D)             // keep polling status
             } while ( x & 0x80 )                // until transfer is done
             SL11write (base_3a + 2, remainder ) // Set data length
             x = SL11read (base_3a)              // get control register
             SL11write ( base_3a, x | 1 )        // and arm it
      }
```

Note that the application is responsible for keeping track whether the remainder data is in EndPoint 3-a or 3-b. In this example, the endpoint is in 3-a.

## 5.18- SL11: AUTO-INCREMENT MODE:

SL11 has supported an 'auto-increment' mode, which the address is written with A0 =0, and each subsequent with A0 = 1 goes to the next higher address in the SL11. It’s is not support multiplexed addresses and data buses.

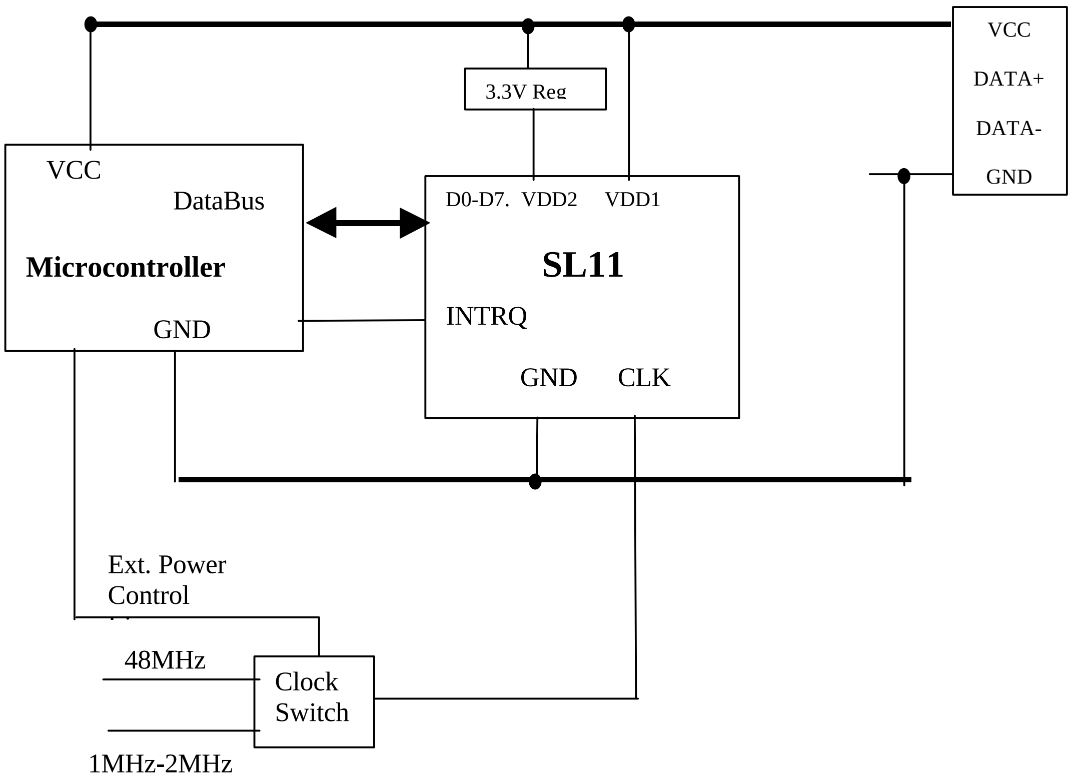
**Note:** Look at **section 6.1** for multiplexed mode (**ALE**).

Here is a part of SL11MemTest function within the SL11.C for auto-increment mode. It starts at 40h up to C0h, which is total SL11 Memory size.

```
      // auto: addr = data
         for (i=EP0Buf, outportb(SL11_ADDR,EP0Buf); i<sMemSize; i++)
             outportb(SL11_ADDR+1,i);

         // auto: addr = data
         for (i=EP0Buf, outportb(SL11_ADDR,EP0Buf); i<sMemSize; i++)
         {
            if ((BYTE)i != (BYTE)inportb(SL11_ADDR+1))
               errors++;
         }
```

## 5.19- APPLICATION NOTE FOR SUSPEND AND RESUME



### 5.19.1- HARDWARE REQUIREMENT FOR SUSPEND AND RESUME:

- The power consumption of the Microcontroller and SL11 can not be more than 500 µA during the suspend mode.
- Requires low power consumption 3.3 regulator.
- Requires a slower clock switch from 48MHz to 1MHz-2MHz range.
- Microcontroller must be able to reduce the processor clock to save the power consumption.

### 5.19.2- SOFTWARE REQUIREMENT FOR SUSPEND AND RESUME:

- To detect the suspend mode, the Firmware must be check the SOF bit5 of the Interrupt status register at address = 0x0D. Within 3 mili-seconds of not seeing the SOF bit, the firmware must switch to slower clock for SL11 and allow the CPU enter the power saving state.

- To detect the Resume mode, the CPU will receive an interrupt from the SL11 to indicate SOF package receive, the firmware must switch back to 48MHz and setup the CPU to full operating power.
  **Note:** The Register 0x05 must have bit 5 enable, in order to receive an interrupt.