SL-11 USB Controller

Technical Reference

OVERVIEW

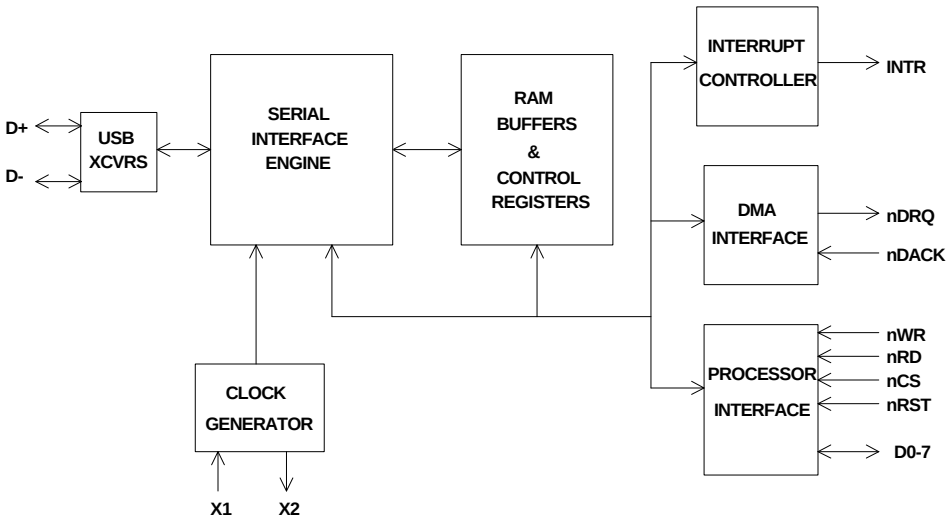
The SL-11 USB Controller is a single chip USB peripheral device solution that enables interfacing to devices such as microprocessors and scanners. The SL-11 USB Controller incorporates USB Serial Interface functionality along with internal drivers and receivers that connect directly to the USB interface connector. The SL-11 supports and operates in USB full speed mode at 12 MBits per second.

The host interface provides an 8 bit data path, with interrupt and DMA support to allow easy interface to standard microprocessors or microcontrollers. Internally, the SL-11 contains a 256 byte RAM data buffer which is utilized for control registers and data storage.

The SL-11 USB Controller is designed to conform with USB specification v1.0 for full speed operation. The USB specification should be used as a reference when designing with the SL-11.

FEATURES

- Standard Microprocessor Interface
- Supports DMA Transfers
- 8 bit Bi-directional Parallel Interface
- 256 x 8 "On-Chip" memory array
- Four USB endpoints
- On-Chip USB transceivers
- Supports power suspend mode
- 5V, 0.8 micron CMOS Technology
- 28 Pin PLCC package
- Generic WDM Mini Port driver for Windows 95/98 & NT, firmware and system USB demo source examples are available.



SL-11 FUNCTIONAL BLOCK DIAGRAM

CONTENTS

SECTION 1
INTRODUCTION
MICROPROCESSOR INTERFACE

The SL-11 microprocessor interface provides an 8 bit bi-directional data path along with appropriate control lines to interface to processors or controllers. The control lines, Chip Select, Read and Write input strobes and a single address line, A0, along with the 8 bit data bus, support programmed I/O or Memory mapped I/O designs.
Access to memory and control register space is a simple two step process, requiring an address write with A0 set = '0' followed by a register/memory read or write cycle with address line A0 set = '1'.
In applications where multiplexed address/data bus support is required, the SL-11 utilizes an ALE (Address Latch Enable) control input to de-multiplex the data bus by latching the register/memory address from the 8 bit data bus when the ALE input strobe is high. Data is written to, or read from the SL-11 with chip select and the appropriate control strobe asserted. In multiplexed applications A0 is not used and must be tied to VDD1.
Data transfers from/to the CPU bus can be accessed at above 10 MBytes/sec rate.

DMA CONTROLLER

In applications which require transfers of large amounts of data, such as scanner interfaces, the SL-11 provides a DMA interface. This interface supports DMA write transfers to the SL-11 internal ram buffer through the microprocessor data bus. Two control lines nDRQ (Data Request) and nDACK (Data Acknowledge) along with the nWrite line control the data flow into the SL-11. The SL-11 has a count register which allow programmable block sizes to be selected for DMA transfer. The control signal interface, both nDRQ and nDACK, are designed to be compatible with standard DMA interfaces. DMA transfers (reads) from the SL-11 out to the 8 bit data bus are not supported.

INTERRUPT CONTROLLER

The SL-11 interrupt controller provides a single output signal, INTRQ which can be activated by a number of events that may occur as a result USB activity. Control and status registers are provided to allow the user to select a single, or multiple events, which will generate an interrupt (assert INTRQ) , and provides a means of viewing interrupt status. Interrupts can also be cleared by writing to the appropriate register.

BUFFER MEMORY

The SL-11 contains 256 bytes of internal buffer memory. The first 64 bytes of the memory represent control and status registers for programmed I/O operations. The remaining memory locations are used for buffering data.
Access to the registers and data memory is through the microprocessor interface 8 bit data bus in either of two addressing modes, indexed or , if used with multiplexed address/data bus interfaces, direct access. With indexed addressing, the address is first written to the device with the A0 address line low, then the following cycle with A0 address line high, is directed to the specified address. In multiplexed address/data bus applications, a single cycle beginning with ALE asserted high latches the address internally, then the read or write strobe accesses the specified location. Address A0 is not used in multiplexed address/data bus interfaces and must be tied to VDD1.
USB transactions are automatically routed to the memory buffer. Control registers are provided, so that the user has the ability to set up pointers and block sizes in buffer memory.

CLOCK GENERATOR

A 48 Mhz external crystal may be used with the SL-11. Two pins, X1 and X2, are provided to connect a lower cost crystal circuit to the device. Circuitry is provided to generate the internal clock requirements of the device. If an external 48 Mhz clock is available in the application, it may be used in lieu of the crystal circuit by connecting directly to the X1 input pin.

USB TRANSCEIVER

The SL-11 has a built in transceiver which meets the USB (Universal Serial Bus) specification v1.0. The transceiver is capable or transmitting and receiving serial data at the USB full speed, 12 Mbits/sec, data rate. The driver portion of the transceiver is differential, while the receive section comprises of a differential receiver and two single ended receivers. Internally, the transceiver interfaces to the SIE (Serial Interface Engine) logic. External, the transceiver connects to the physical layer of the USB.

SL-11 REGISTERS

Operation of the SL-11 is effected through the SL-11 internal register set. A portion of the internal ram is devoted to the register space and access is provided through the microprocessor interface. The registers provide control and status information for transactions on the USB, microprocessor interface, DMA and interrupts.

POWER RESUME AND SUSPAND MODE

SL11 has build in SOF Interrupt signal that can be monitored by an external microprocessor , SOF indicates continuous USB activity and it is inserted every one (1) msec. User processor can use a counter to count 3-5 msec of non-SOF's, non-USB activity, and thus start power Suspend sequence. The peripheral requires additional hardware that provides the abilities to shut power to the unit, not 3.3v power to USB Data+ pull-up resistor, and turn-off 48Mhz clock to the SL11, then the unit can be suspended and draw less then 100 microA. Also, CPU firmware needs to take in consideration protecting stack, flags and to remember that it went to suspend mode.

Power resume sequence can be achieved by enabling an external wake up interrupt line to the CPU/IRQ, to the external processor, utilizing Data+ transition that indicates the resumption of USB activity. For further detail, please contact ScanLogic personnel to provide you with additional detail information and application note.

SECTION 2
SL-11 REGISTERS

The registers in the SL-11 are divided into two major groups. The first group, known as Endpoint Registers, are involved in USB control transactions and data flow. The second group, referred to as the USB Registers, provide the control and status information for all other operations.

ENDPOINT REGISTERS

Communication and data flow on the USB is implemented using endpoints. These uniquely identifiable entities are the terminals of communication flow between a USB host and USB devices. Each USB device is composed of a collection of independently operating endpoints. Each endpoint has a unique identifier: the Endpoint Number. See USB specification v1.0. Sec 5.3.1.
The SL-11 supports four endpoints, numbered 0 - 3. Endpoint 0 is the default pipe, and is used to initialize and manipulate the device. It also provides access to the device's configuration information, and supports control transfers. Endpoints 1 - 3 can support bulk, interrupt, or Isochronous transfers.
Each endpoint has two sets of registers, 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 SL-11 is ready for the next transfer without interruption.

ENDPOINT REGISTER ADDRESSES

Each endpoint set has a group of five registers which are mapped in the SL-11 memory. The register sets have address assignments as shown in the following table.

Endpoint Register Set	Address (in Hex)
Endpoint 0 - a	00 - 04
Endpoint 0 - b	08 - 0C
Endpoint 1 - a	10 - 14
Endpoint 1 - b	18 - 1C
Endpoint 2 - a	20 - 24
Endpoint 2 - b	28 - 2C
Endpoint 3 - a	30 - 34
Endpoint 3 - b	38 - 3C

For each endpoint set (starting at address xx) the registers are mapped as shown in the following table:

Endpoint Register Set (for Endpoint *n* starting at register position *xx*)	
xx	Endpoint *n* Control
xx +1	Endpoint *n* Base Address
xx +2	Endpoint *n* Base Length
xx +3	Endpoint *n* Packet Status
xx +4	Endpoint *n* Transfer Count

Endpoint Control Register:

Each endpoint set has a control register defined as follows:

Bit Position	Bit Name	Function
0	Arm	Allows enabled transfers when set =’1’. Cleared to '0' when transfer is complete.
1	Enable	When set = '1' allows transfers to this endpoint. When set "0 USB transactions are ignored. If Enable = '1' and Arm = '0' the endpoint will return NAK's to USB transmissions.
2	Direction	When set = '1' transmit to Host. when '0' receive from Host.
3	Next Data Set	'0' if next data set is 'a' , '1' if next data set is 'b'.
4	ISO	When set to '1' allows isochronous mode for this endpoint.
5	Send Stall	When set to '1' sends Stall in response to next request on this endpoint.
6	Sequence	Sequence Bit. '0' if DATA0, '1' if DATA1.
7	Not Defined	

Endpoint Base Address:

Pointer to memory buffer location for USB reads and writes.

Endpoint Base Length :

Maximum packet size for Out transfers from Host. Essentially, this designates the largest packet size that can be received by the SL-11. For Transfers In to Host, Base Length designates the size of data packet to send.

Packet Status:

The packet status contains information relative to the packet which has been received or transmitted. The register is defined as follows:

Bit Position	Bit Name	Function
0	Ack	Transmission Acknowledge.
1	Error	Error detected in transmission.
2	Time-out	Time-out occurred.
3	Sequence	Sequence Bit. '0' if DATA0, '1' if DATA1.
4	Setup	'1' indicates Setup Packet
5	Overflow	Overflow condition - maximum length exceeded during receive.
6	Not Defined	
7	Not Defined	

Transfer Count:

Contains the number of bytes left over (from 'Length' field) after a packet is transferred. If an overflow condition occurs, i.e., the received packet from host was greater than the Length field, a bit is set in the Packet Status Register indicating the condition.

USB REGISTERS

In addition to the Endpoint Registers, the SL-11 contains a number of other registers which provide control and status functions.

Register Address Assignments:

The control and status registers are mapped as follows:

Register Name	Address (in Hex)
Control Register	05 H
Interrupt Enable Register	06 H
USB Address Register	07 H
Interrupt Status Register	0D H
Current Data Set Register	0E H
SOF Low Byte Register	15 H
SOF High Byte Register	16 H
DMA Total Count Low Byte Register	35 H
DMA Total Count High Byte Register	36 H

Control Register, Address 05H:

The Control Register enables/disables USB transfers and DMA operation with control bits defined as follows:

Bit Position	Bit Name	Function
0	USB Enable	Overall Enable for transfers. '1' enables, '0' disables.
1	DMA Enable	Enable DMA transfer. '1' enables, '0' disables.
2-7	Reserved	Reserved bit - all must be set to '0' 's.

Interrupt Enable Register, Address 06H:

The SL-11 provides an Interrupt Request Output which can be activated on a number of conditions. The Interrupt Enable Register allows the user to select activities that will generate the Interrupt Request. A separate Interrupt Status Register is provided. It can be polled in order to determine the condition which initiated the interrupt. (See Interrupt Status Register description).
When a bit is set to '1' the corresponding interrupt is enabled.

Bit Position	Bit Name	Function
0	EP 0 Done	Enable Endpoint 0 done Interrupt.
1	EP 1 Done	Enable Endpoint 1 done Interrupt.
2	EP 2 Done	Enable Endpoint 2 done Interrupt.
3	EP 3 Done	Enable Endpoint 3 done Interrupt.
4	DMA Done	Enable DMA done Interrupt.
5	SOF Rec'vd	Enable SOF Received Interrupt.
6	USB Reset	Enable USB Reset Interrupt.

Interrupt Status Register, Address 0DH:

This Read/Write register serves as an interrupt status register when read. and an Interrupt clear register when written. To clear an interrupt bit, the register must be written with the appropriate bit set to '1'. Writing a '0' has no affect on the status.
Note: Bit 7 is not an interrupt driven status, when it is clear, during DMA cycle, it indicates that the USB host have taken the data from the SL11. Thus, last DMA transaction was completed and only now it is ok to disable DMA.

Bit Position	Bit Name	Function
0	EP 0 Done	Endpoint 0 done Interrupt.
1	EP 1 Done	Endpoint 1 done Interrupt.
2	EP 2 Done	Endpoint 2 done Interrupt.
3	EP 3 Done	Endpoint 3 done Interrupt.
4	DMA Done	DMA done interrupt, Last byte moved to the DMA buffer (DMA count = 0), but DMA buffer has not yet moved to the USB host, bit 7 needs be checked to find out if DMA data got moved to the USB host.
5	SOF Rec'vd	SOF Received interrupt.
6	USB Reset	USB Reset received interrupt.
7	USB-DMA (not interrupt driven status)	USB-DMA Transfer in progress status. When it is clear , it means that DMA buffers are empty, all DMA data got moved to USB host. Bit 7 is not interrupt driven status bit, user must poll this bit after bit 4 DMA done interrupt bit is set.

USB Address Register, Address 07H:

USB Device Address. Register location where the unique USB Device Address is saved after assignment by USB Host. Initially set = 0 for configuration. After configuration and address assignment, only transactions directed to this address will be recognized by the device.

Current Data Set Register, Address 0EH:

Read Only. Register indicates currently selected data set for each endpoint.

Bit Position	Bit Name	Function
0	Endpoint 0	'0' for Endpoint 0a, '1' for Endpoint 0b.
1	Endpoint 1	'0' for Endpoint 1a, '1' for Endpoint 1b.
2	Endpoint 2	'0' for Endpoint 2a, '1' for Endpoint 2b.
3	Endpoint 3	'0' for Endpoint 3a, '1' for Endpoint 3b.
4-7	Not Defined	

SOF Low Register, Address 15H:

Read Only
Contains low order 7 bits of Frame Number. (bits 7-1).

SOF High Register, Address 16H:

Read Only
Contains high order 4 bits of Frame Number. (bits 7-4).

DMA Total Count Low Register, Address 35H:

Contains low order 8 bits of DMA count.
DMA total count is the total number of bytes to be transferred from a peripheral to the SL-11. The count may sometimes require up to 16 bits, thus the count is represented in two registers, Total Count Low, and Total Count High.

DMA Total Count High Register, Address 36H:

Contains high order 8 bits of DMA count.

SECTION 3
DMA OPERATION

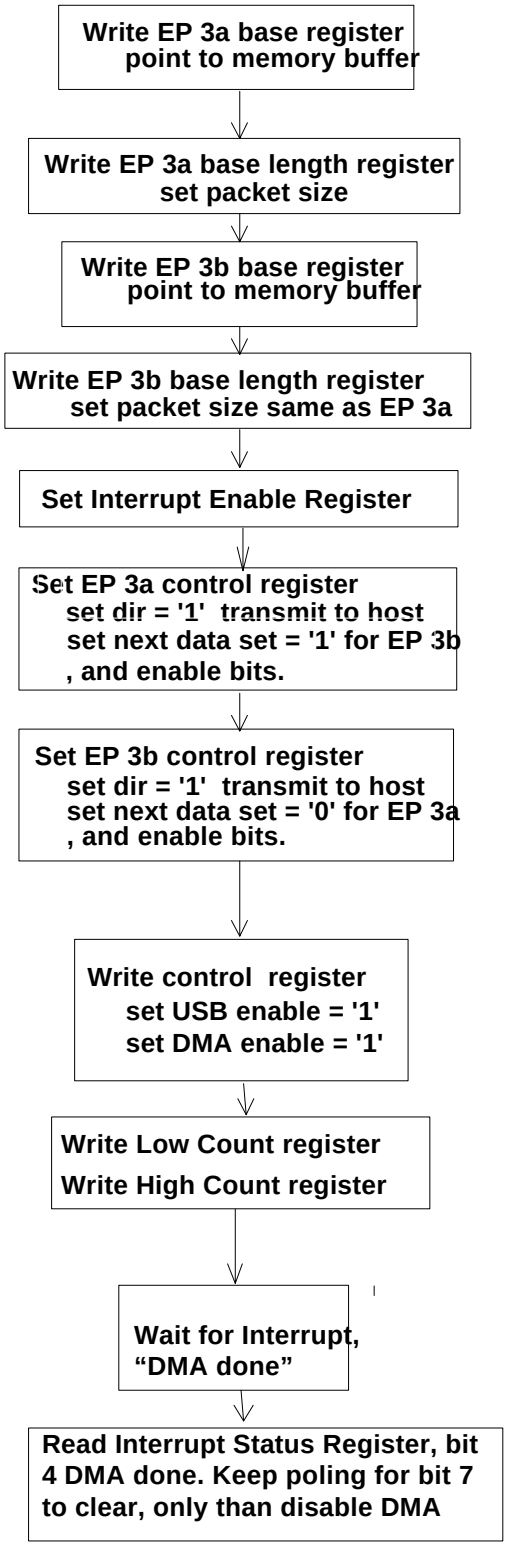
The SL-11 will support DMA transfers on the 8 bit data bus. Two signals nDRQ and nDACK are used along with the nWR signal to move data into the SL-11 from the microprocessor data bus. Note: DMA read transfers from the SL-11 to the microprocessor bus are currently not supported.

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 opposites 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.

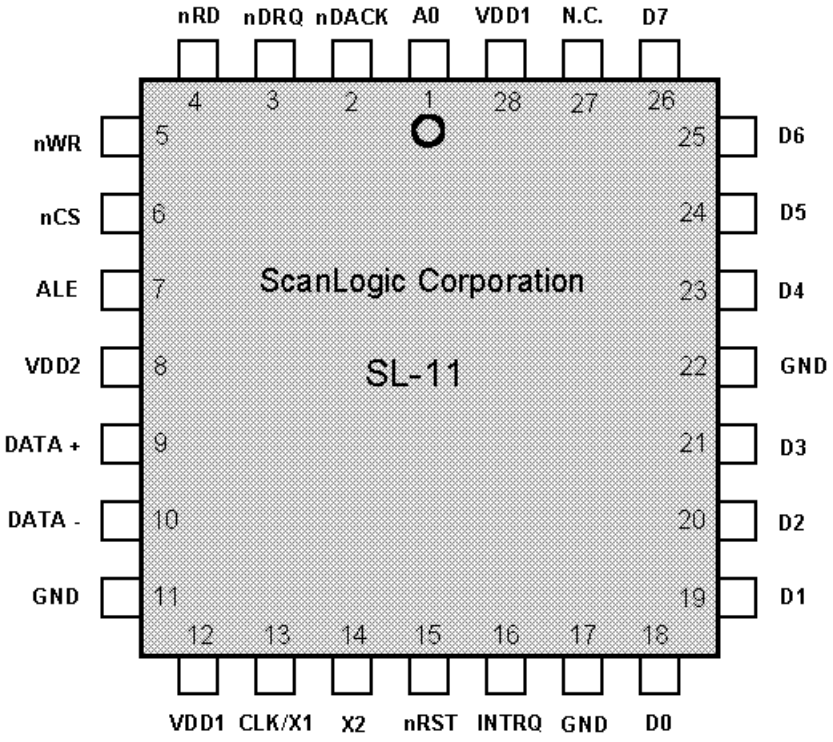
DMA SETUP AND OPERATION

The following flowchart represents a typical DMA setup and sequence of operation:



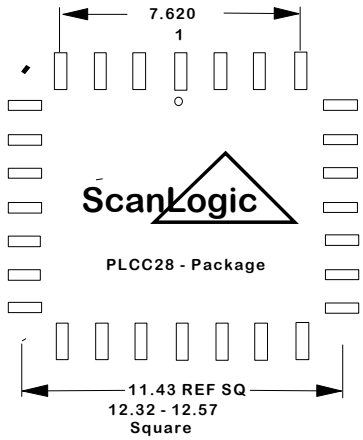
SECTION 4
SL-11 PIN INFORMATION

The diagram below indicates the pin assignments for the SL-11 USB Controller.



SL-11 PIN DIAGRAM

The SI-11 is available in a 28 pin PLCC.



PIN DESCRIPTIONS

The SL-11 is packaged in a 28 Pin PLCC. The device requires a +5 VDC and a +3.3 VDC supply, average typical current consumption is less then 30 ma (for both 5/3.3v). The SL-11 requires an external 48 MHz crystal or Clock.

SL11 pin assignments and definitions are set forth in the following table.

Pin No.	Pin Type	Pin Name	Pin Description
1	IN	A0	A0 = '0'. Selects Addr. Pointer. Reg. Write Only. Note * A0 = '1'. Selects Data Buffer or Register. R/W. In Multiplexed address applications, AO should be tied to VDD1.
2	IN	nDACK	DMA Acknowledge. An active low input used to interface to an external DMA controller.
3	OUT	nDRQ	DMA Request. An active low output used with an external DMA controller. nDRQ and nDACK form the handshake for DMA data transfers.
4	IN	nRD	Read Strobe Input. An active low input used with nCS to read registers/data memory.
5	IN	nWR	Write Strobe Input. An active low input used with nCS to write to registers/data memory, or with nDACK during DMA transfers to write to memory.
6	IN	nCS	Active low SL-11 Chip select. Used with nRD and nWr when accessing SL-11. nCS must be held inactive during a DMA cycle.
7	IN	ALE	Address Latch Enable. Used to De-Multiplex 8 bit Address/Data Bus. Note **
8	VDD2	3.3 VDC	Power for USB Transceivers. VDD2 may be derived from VDD1. Note ****
9	BIDIR	DATA +	USB Differential Data Signal High Side.
10	BIDIR	DATA -	USB Differential Data Signal Low Side.
11	GND	USB GND	Ground Connection for USB.
12	VDD1	+5 VDC	SL-11 Device VDD Power.
13	IN	CLK/X1	48 Mhz Clock or External Crystal X1 connection.
14	OUT	X2	External Crystal X2 connection.
15	IN	nRST	SL-11 Device active low reset input.
16	OUT	INTRQ	Active high Interrupt Request output to external controller.
17	GND	GND	SL-11 Device Ground.
18	BIDIR	D0	Data 0. Microprocessor Data/(Address) Bus. ***
19	BIDIR	D1	Data 1. Microprocessor Data/(Address) Bus. ***
20	BIDIR	D2	Data 2. Microprocessor Data/(Address) Bus. ***
21	BIDIR	D3	Data 3. Microprocessor Data/(Address) Bus. ***
22	GND	GND	SL-11 Device Ground.
23	BIDIR	D4	Data 4. Microprocessor Data/(Address) Bus. ***
24	BIDIR	D5	Data 5. Microprocessor Data/(Address) Bus. ***
25	BIDIR	D6	Data 6. Microprocessor Data/(Address) Bus. ***
26	BIDIR	D7	Data 7. Microprocessor Data/(Address) Bus. ***
27	NC	No Connect	Pin must be left unconnected.
28	VDD1	+5 VDC	SL-11 Device VDD Power.

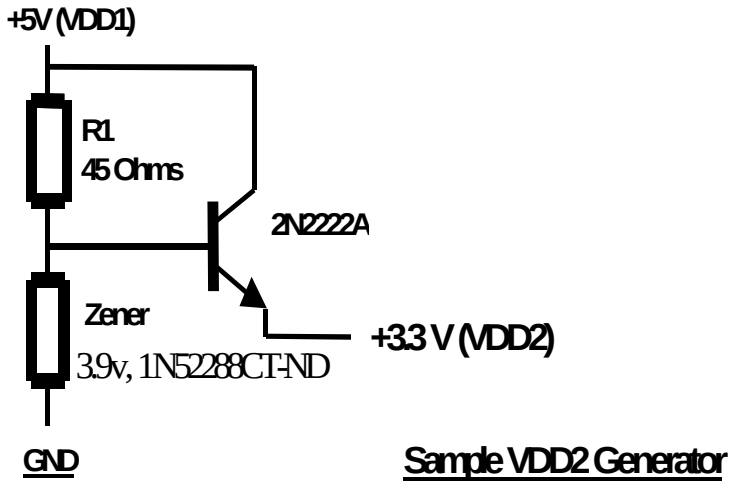
Notes:

* A0 Address bit is used to access address register or data registers in I/O Mapped or Memory Mapped applications. If SL-11 interface is to a multiplexed address/data bus, A0 should be tied to VDD1.

** ALE Address Latch Enable signal is used only with MicroController buses which require de-multiplexing Address and Data from a single 8 Bit bus. Normally must be tied low.

*** D0 - D7 normally is a data bus, but in applications which utilize a multiplexed address/data bus, the bus will contain an address when ALE is asserted high.

**** VDD2 can be derived from the VDD1 supply with a few additional components.
The Diagram below illustrates a simple method which will provide 3.3V, up to 30mA.



Sample VDD2 Generator

***** X1/X2 Clock requires external 48MHz matching crystal or clock source.

SECTION 5
ELECTRICAL SPECIFICATIONS

ABSOLUTE MAXIMUM RATINGS

This section list the absolute maximum ratings of the SL-11. Stresses above those listed can cause permanent damage to the device. Exposure to maximum rated conditions for extended periods can affect device operation and reliability.

Storage Temperature	-40 to 125^0 C
Voltage on any pin with respect to ground	-0.3v to VDD1 + 0.3V
Power Supply Voltage (VDD1)	5 V ± 10%
Power Supply Voltage (VDD2)	3.3V ± 10%
Lead Temperature (10 seconds)	*TBD*

RECOMMENDED OPERATING CONDITIONS

Parameter	Min.	Typ.	Max
Power Supply Voltage, VDD1	4.75 V		5.25 V
Power Supply Voltage, VDD2	3.0 V		3.6 V
Operating Temperature	0^0 C		65^0 C

Crystal Requirements, (XTAL1, XTAL2)	Min.	Typ.	Max
Operating Temperature Range	0^0 C		65^0 C
Parallel Resonant Frequency		48 Mhz	
Frequency Drift over Temperature			+/- 20 ppm
Accuracy of Adjustment			+/- 30 ppm
Series Resistance			50 ohms
Shunt Capacitance	3 pf		6 pf

The device requires +5 VDC and a +3.3 VDC supply, average typical current consumption is less then 30 ma for both 5/3.3v, and it less then 80 uA when clk is disabled.

EXTERNAL CLOCK INPUT CHARACTERISTICS (XTAL1)

Parameter	Min.	Typ.	Max
Clock Input Voltage @ XTAL1 (XTAL2 Open)	1.5 V		
Clock Frequency		48 Mhz	

SL-11 DC CHARACTERISTICS (*PRELIMINARY*)

Symbol	Parameter	Min.	Typ.	Max
V_{IL}	Input Voltage LOW	-0.5 V		0.8 V
V_{IH}	Input Voltage HIGH	2.0V		VDD1+ 0.3V
V_{OL}	Output Voltage LOW			0.6 V
V_{OH}	Output Voltage HIGH	2.4 V		
C_{IN}	Input Capacitance			20 pf
I_{CC}	Supply Current (VDD1)			20ma
I_{USB}	Supply Current (VDD2)			10ma

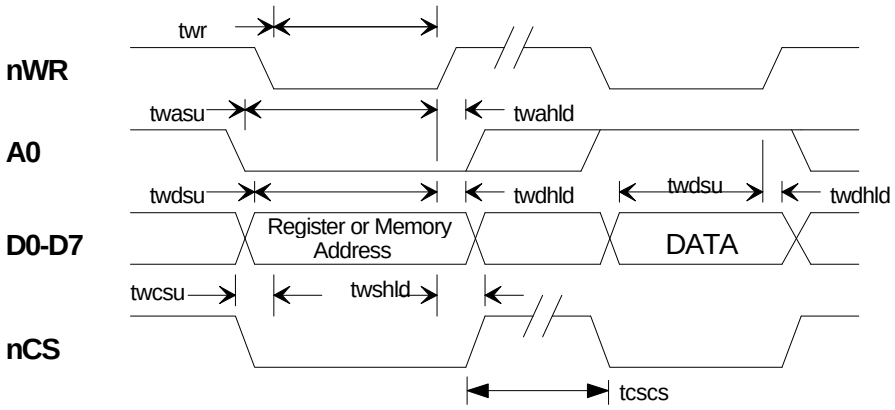
SL-11 USB TRANSCEIVER CHARACTERISTICS

Symbol	Parameter	Min.	Typ.	Max
V_{IHYS}	Hysteresis On Input (Data+, Data-)	0.1 V		200 mV
V_{USBIH}	USB Input Voltage HIGH		1.5 V	2.0 V
V_{USBIL}	USB Input Voltage LOW	0.8 V	1.3 V	
V_{USBOH}	USB Output Voltage HIGH	2.2 V		
V_{USBOL}	USB Output Voltage LOW			0.7 V
Z_{USBH}	Output Impedance HIGH STATE	24 Ohms		28Ohms
Z_{USBL}	Output Impedance LOW STATE	24 Ohms		28 Ohms
I_{USB}	Transceiver Supply p-p Current (3.3V)	100mA		

Notes:
1. All typical values are VDD2 = 3.3 V and TAMB= 250 C.
2. Z_{USBX} Impedance Values includes an external resistor of 24 Ohms +/- 1%.

SL-11 BUS INTERFACE TIMING REQUIREMENTS

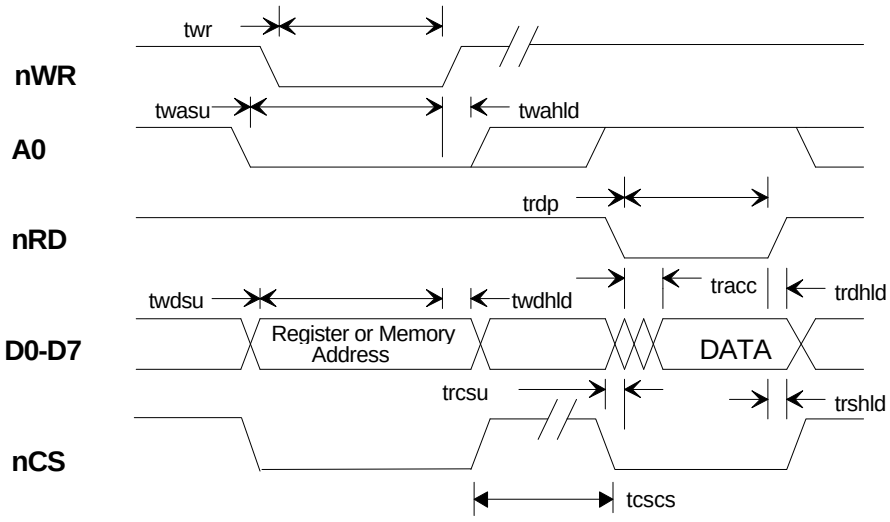
I/O WRITE CYCLE



I/O Write Cycle to Register or Memory

Symbol	Parameter	Min.	Typ.	Max
twr	Write pulse width	65 nsec		
twcsu	Chip select setup to nWR	10 nsec		
twshld	Chip select hold time	10 nsec		
twasu	A0 address setup time	65 nsec		
twahld	A0 address hold time	10 nsec		
twdsu	Data to write low setup time	5 nsec		
twdhld	Data hold time after write high	10 nsec		
tcscs	nCS inactive to nCS asserted	85 nsec		

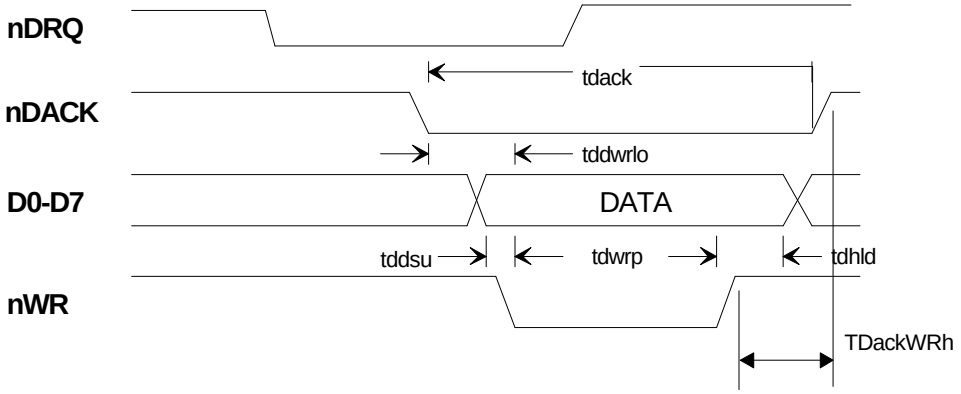
I/O READ CYCLE



I/O Read Cycle from Register or Memory Buffer

Symbol	Parameter	Min.	Typ.	Max
twr	Write pulse width	65 nsec		
trd	Read pulse width	65 nsec		
twasu	Chip select setup to nWR	10 nsec		
twasu	A0 address setup time	75 nsec		
twahld	A0 address hold time	10 nsec		
twdsu	Data to write high setup time	5 nsec		
twdhld	Data hold time after write high	10 nsec		
tracc	Data valid after read low	20 nsec		25 nsec
trdhld	Data hold after read high	5 nsec		
trcsu	Chip select low to read low	10 nsec		
trshld	Chip select hold after read high	10 nsec		
tcscs	nCS inactive to nCS asseted	85 nsec		

DMA Write CYCLE

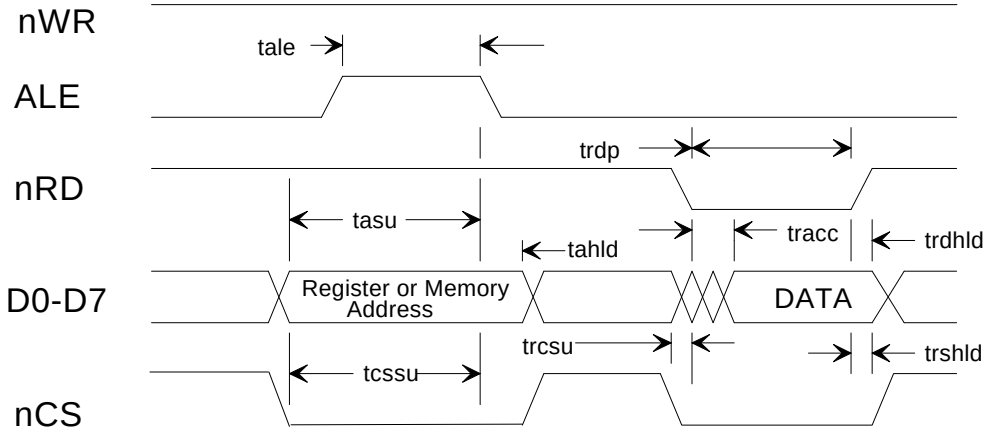


SL-11 DMA WRITE CYCLE TIMING

Symbol	Parameter	Min.	Typ.	Max
tdack	nDACK low	35nsec		
tddwrlo	nDACK to nWR low delay	10 nsec		
tdackh	nDRQ high to nDACK high delay	45 tbd	40 nsec	
tdwrp	nWR pulse width	25 nsec		
tdhld	Date hold to nWR high	7.5 nsec		
tddsu	Data setup to nWR strobe low	5 nsec		
tddackwrh	nWR high to nDACK high	5 nsec		

Note: nWR must go low after nDACK goes low in order for nDRQ to clear. If this sequence is not implemented as requested, the next nDRQ will be not inserted.

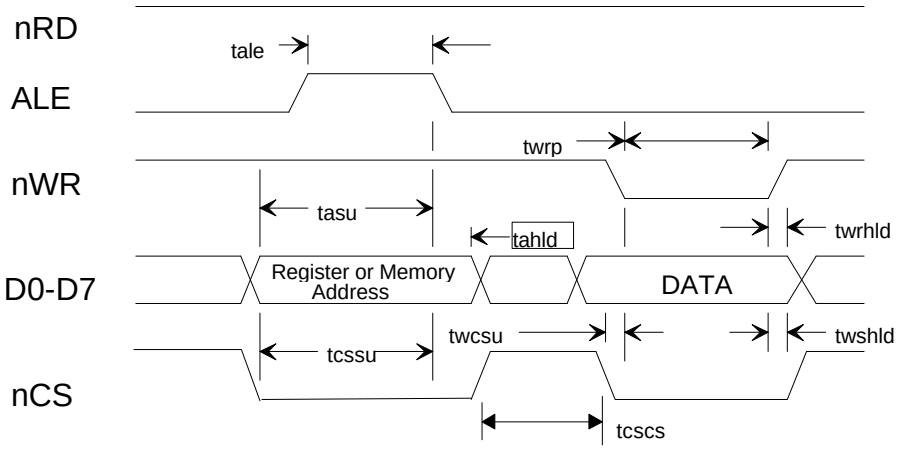
Multiplexed Address/Data Bus Read Cycle



Multiplexed Address I/O Read

Symbol	Parameter	Min.	Typ.	Max
tale	ALE pulse width	65 nsec		
trdp	nRd low pulse width	65 nsec		
tasu	Address setup to ALE low	65 nsec		
tahld	Address hold after ALE low	10 nsec		
tracc	Data access after nRD low	20 nsec		80 nsec
trdhld	Data hold after nRD high	5 nsec		
tcssu	nCS low setup to ALE low	15 nsec		
trcsu	nCS low to nRD low	10 nsec		
trshld	nCS hold after nRD high	10 nsec		

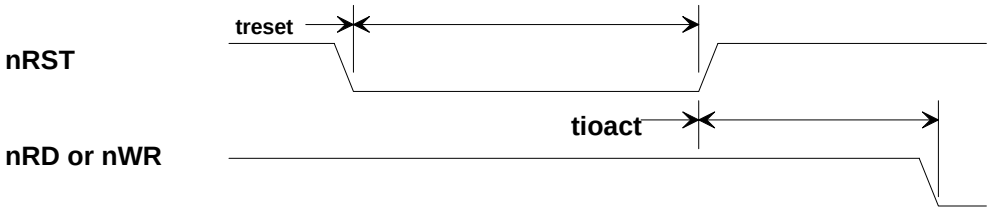
Multiplexed Address/Data Bus Write Cycle



Multiplexed Address I/O Write

Symbol	Parameter	Min.	Typ.	Max
tale	ALE pulse width	65 nsec		
twrp	nWR low pulse width	65 nsec		
tasu	Address setup to ALE low	65 nsec		
tahld	Address hold after ALE low	10 nsec		
tdsetup	Data valid to nWR low	5 nsec		
twrhld	Data hold after nWR high	10 nsec		
tcssu	nCS low setup to ALE low	55 tbd		
twcsu	nCS low to nWR low	10 nsec		
twshld	nCS hold after nWR high	10 nsec		
tcscs	nCS inactive to nCS asserted	85 nsec		

SL-11 RESET TIMING

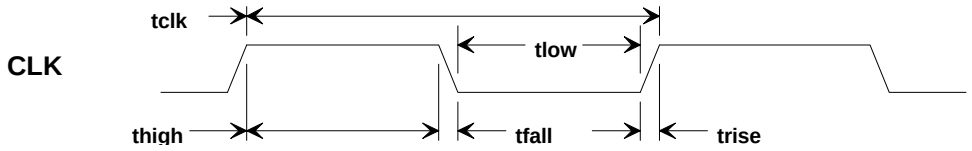


SL-11 RESET TIMING

Symbol	Parameter	Min.	Typ.	Max
treset	nRst Pulse width	16 clocks		
tioact	nRst high to nRD or nWR active	16 clocks		

Note: Clock is 48 MHz nominal.

SL-11 Clock Timing Specifications

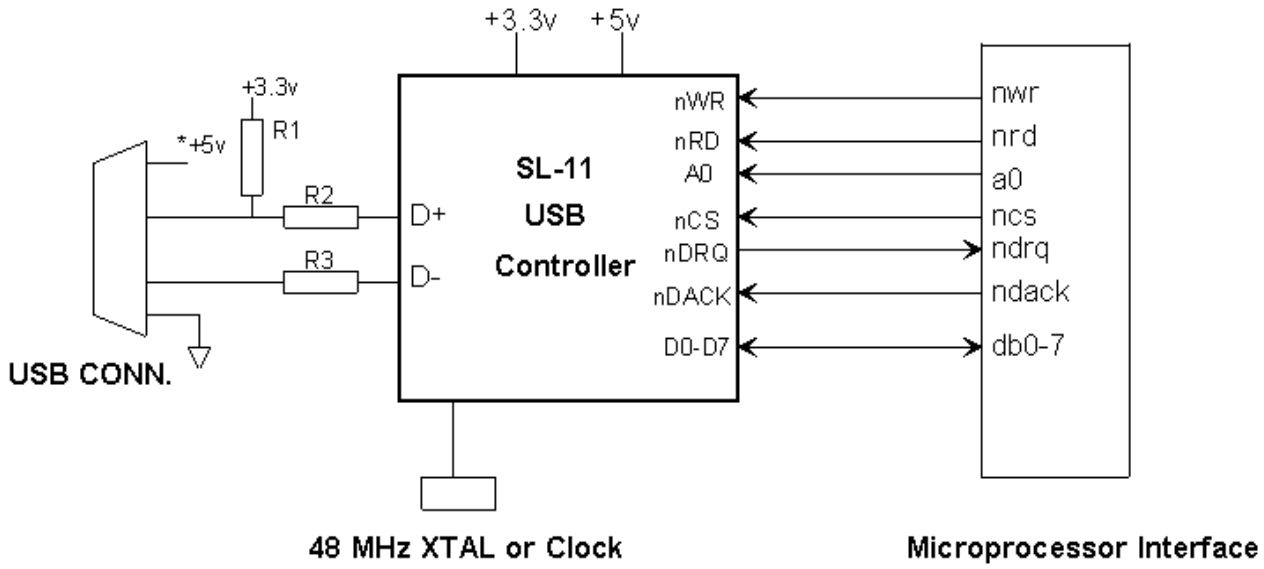


SL-11 CLOCK TIMING

Symbol	Parameter	Min.	Typ.	Max
tclk	Clock period (48 Mhz)	20.0 nsec	20.8 nsec	
thigh	Clock high time	9 nsec		11 nsec
tlow	Clock low time	9 nsec		11 nsec
trise	Clock rise time			5.0 nsec
tfall	Clock fall time			5.0 nsec
	Duty Cycle	-5%		+5%

SL-11 CIRCUIT APPLICATION

A typical application for the SL-11 USB Controller requires a minimum of external components. The Block Diagram shown below illustrates such an application to interface to a standard microprocessor.



R1 pull-up terminator is a 1.5K resistor +/- 5% and is required for full speed devices.

(See USB Specification v1.0 Section 7.1.3).

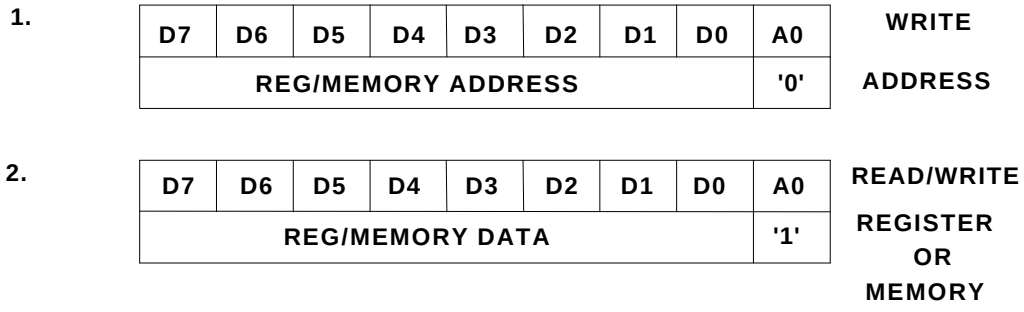
R2 and R3 are typically 24 Ohms.

(See USB Specification v1.0 Section 7.1.1).

*** +5v supplied by USB is capable of sourcing 500 ma max.**

BLOCK DIAGRAM - TYPICAL SL-11 - MICROPROCESSOR APPLICATION

SECTION 6
PROGRAMMING INFORMATION
DEVICE PROGRAMMING



SL-11 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 SL-11 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.

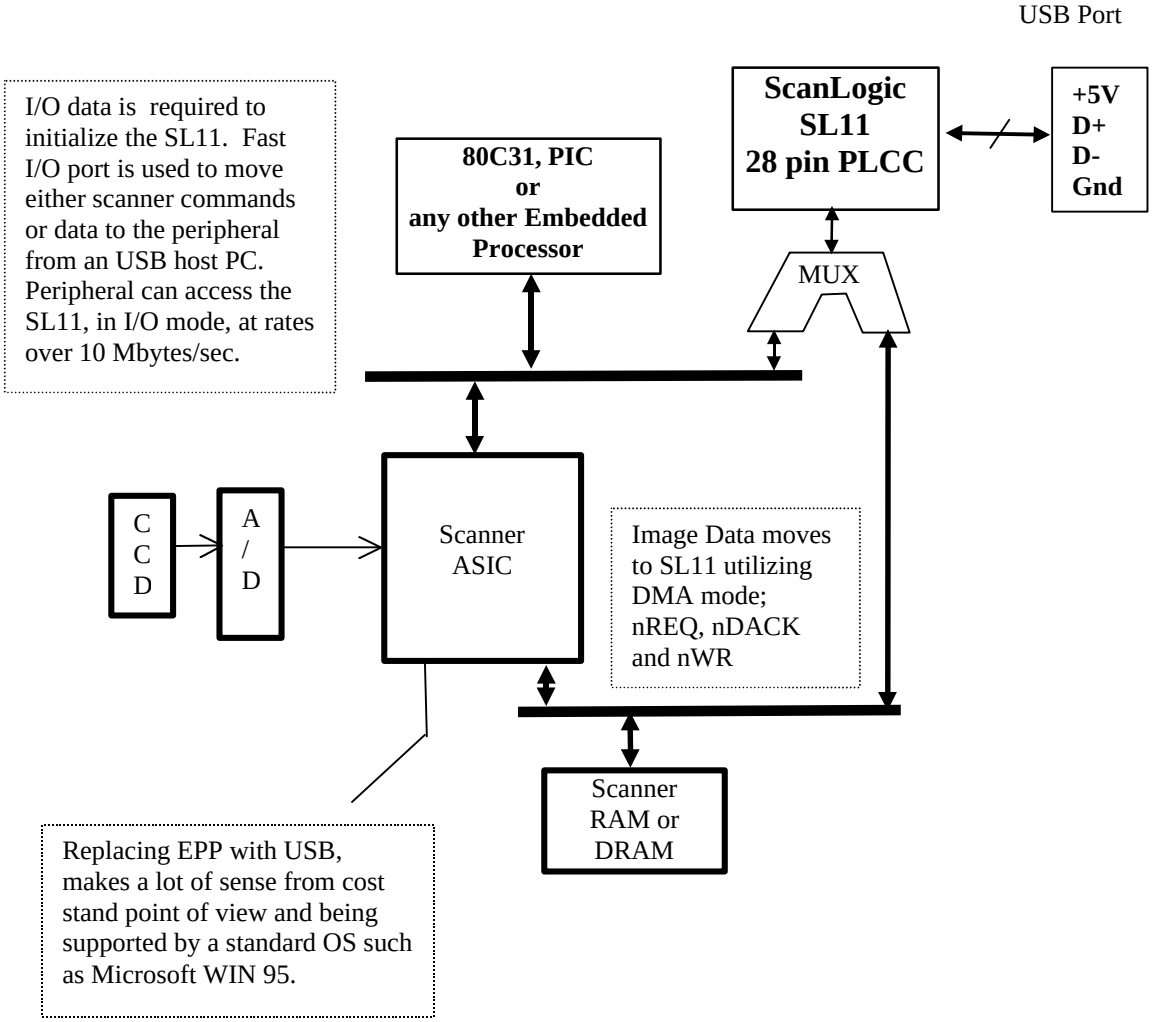
PROGRAMMING ENDPOINTS

The SL-11 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 SL-11 is ready for the next transfer without interruption.

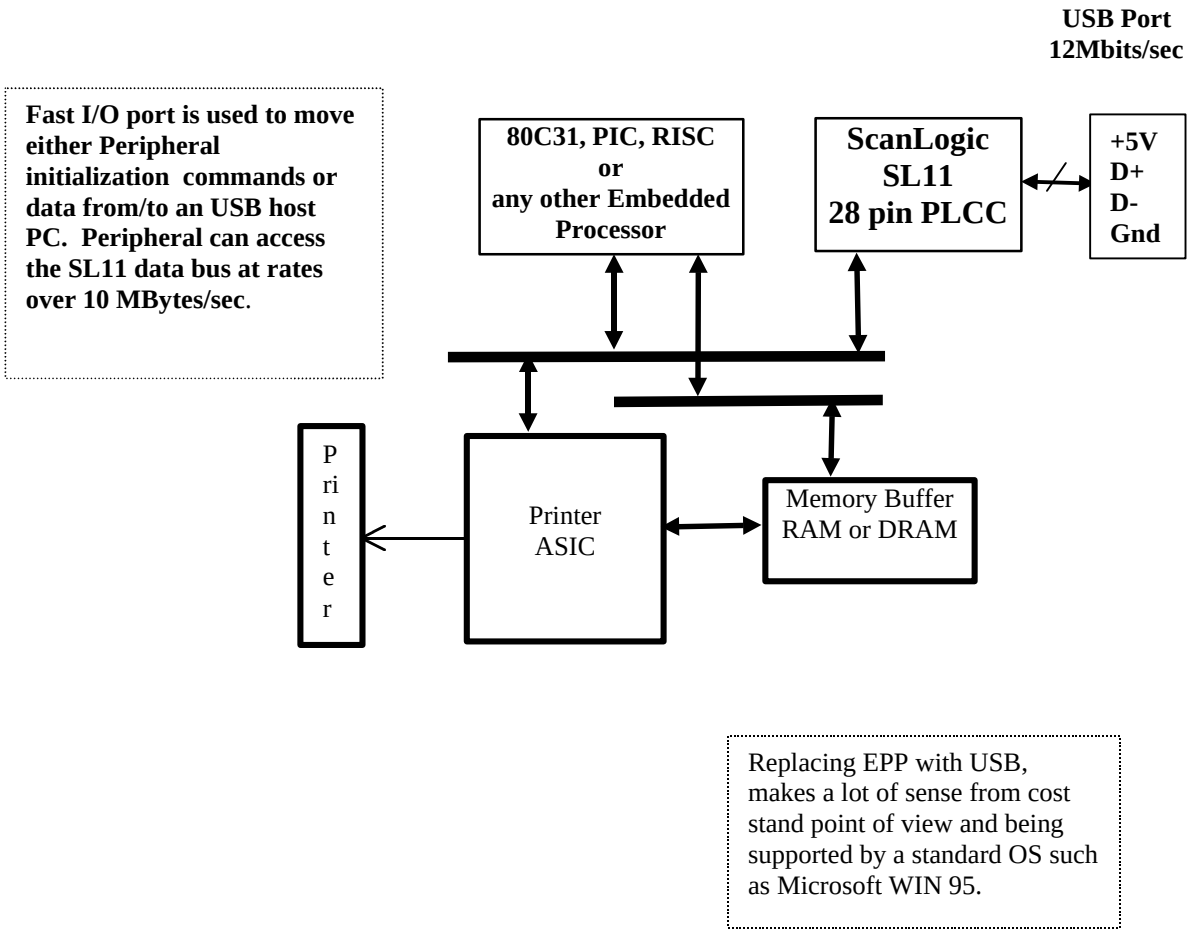
Each endpoint has an associated set of registers, described in section 4, which must be programmed in order initiate or respond to transactions on the USB. Endpoint EP0 transactions are initiated by the host during setup and configuration. Typically the host will request information from the device during 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.

Once configuration has been completed, the device can be programmed to send or receive data using EP1 - 3.

SECTION 7
APPLICATION NOTES INFORMATION
DEVICE FIRMWARE PROGRAMMING AND DEMO SW;



ScanLogic Corporation – SL11 USB Interface Chip
Scanner Application Block Diagram



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

SECTION 8
SL-11 APPLICATION NOTE #SL11-01

OVERVIEW

The purpose of this document is to provide a basic guide to programming the SL-11 USB Controller.
The next section will illustrate how a microcontroller might communicate with the SL-11. In successive sections, examples will be shown for three possible operations to be performed with the SL-11:

- Transmit Data
- Receive Data
- DMA Transfer

TALKING TO THE SL-11

The exact hardware interface used to communicate with the SL-11 is beyond the scope of this document, and can vary depending on what device the SL-11 is connected to. This section describes the process only in general terms.
Writing a byte to the SL-11 involves two writes cycles: First, the application must write a register address into the SL-11's Address Pointer Register; Second, the actual data is written to the chip in a second write.
Reading a byte from the SL-11 involves a write cycle followed by a read cycle: First, as in a data write, the application writes a register address into the SL-11'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 SL-11 is used to select the Address Register or the Data register. In microcontrollers that multiplex the address and data buses, the ALE pin is used to demultiplex the address and data buses. For a detailed description of the hardware interface, please refer to the SL-11 USB Controller Specification.
In an I/O mapped application, a function to write a byte to an SL-11 register might read like this:

```
write ( register, data )
{
 out ( addr, register )        // 'addr' is the SL-11's base address
 out ( addr + 1, data )
}
```

And a function to read an SL-11 register might look like this:

```
read ( register )
{
 out ( addr, register )
 return_value =  in ( addr + 1 )
}
```

TRANSMIT DATA TO USB HOST

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

SENDING A PACKET

Assuming that the data we want to send is already loaded into the SL-11's buffer, starting at address 68 hex, the sequence would be:

```
base = 0x10             //  hex 10 is base register for endpoint 1-a
write ( base+1, 0x68 )  //  load base address register with data addr
write ( base+2, 8 )     //  load data length into length register
x = read ( 0x06 )       //  get contents of interrupt enable register
write ( 0x06, x | 2 )   //  OPTIONAL: enable endpoint 1 done interrupt
write ( 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.

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 = read ( 0x0D )             // read the interrupt status register
if ( x & 2 )                  // interrupt caused by Endpoint 1 done
{
      status = read ( 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
}
```

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 = read ( base ) // get the contents of the control register
      write ( base, x | 1 )   // reset the ‘armed’ bit
}
```

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.

RECEIVING A PACKET

Assuming that we want the data to be placed in the SL-11's buffer starting at address 60 hex, the sequence would be:

```
base = 0                // this is the base address for endpoint 0-a
write ( base + 1, 0x60 ) // buffer address into which to place data
write ( base + 2, 0x40 ) // maximum length of data to be received
x = read ( 0x06 )       // get contents of interrupt enable register
write ( 0x06, x | 1 )   // OPTIONAL: enable Endpoint 0 interrupt
write ( 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.

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 = read ( 0x0D )       // read the interrupt status register
if ( x & 1 )            // yes, interrupt caused by Endpoint 0 done
{
      status = read ( 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 Transfer Count register 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 - read( base + 4 )
```

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

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 = read ( base )       // get the control register contents
      write ( 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.

USING THE "PING-PONG" BUFFERS

The SL-11 provides two sets of registers for each endpoint. They are referred to in this document and in the SL-11 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 SL-11 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 microcontroller in use.

SETTING UP A DMA TRANSFER

In this example, we want to set up a DMA transfer from application memory to the SL-11, with the data to then be sent to the host. In the SL-11, endpoint 3 is used for DMA operations.
This example also uses the SL-11's ping-pong buffers, which were ignored in the first two examples for the sake of simplicity.

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 SL-11'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 = read ( 5 )                  // read control register
write ( 5, x | 2 )              // make sure DMA enable bit is set
write ( base_3a+1, 0x80 )       // data address for EP 3-a
write ( base_3b+1, 0xC0 )       // data address for EP 3-b (EP 3-a + 40h)
write ( base_3a+2, 0x40 )       // data size for EP 3-a  (64 bytes)
write ( base_3b+2, 0x40 )       // data size for EP 3-b  (64 bytes)
write ( 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
write ( 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 = read ( 0x06 )         // get contents of interrupt enable register
write ( 0x06, x | 0x10 ) // enable DMA Done interrupt
write ( 0x35, 0 )         // Low Byte of DMA Total Count
write ( 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 SL-11's DMA transfer process. The SL-11 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 = read ( 0x0D )               // read interrupt status register
if ( x & 0x10 )                 // it's the DMA done interrupt
       do whatever is next
else
       handle other interrupts as needed
```

AN "ODD" DMA TRANSFER

As mentioned in the SL-11 USB Controller Specification, the DMA mechanism will only send a whole number multiple of the packet size set in the SL-11'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 SL-11'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

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

You would need to write:

```
write ( 0x35, 0x0B )    // Low Byte of DMA Total Count
write ( 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 = read ( 0x0D )             // keep polling status
      } while ( x & 0x80 )                // until transfer is done
      write (base_3a + 2, remainder )     // Set data length
      x = read ( base_3a )                // get control register
      write ( 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.

SECTION 9

SL-11 SYSTEM SOFTWARE SPECIFICATION

OVERVIEW

This document is a preliminary specification for the software interface to the SL-11. These functions are part of a USB Host side example to be supplied to developers

REVISION HISTORY

1.00 First draft 10/15/96 EAP

DEFINED TYPES AND DATA STRUCTURES

```
/*****************************************************************************/
/* ScanLogic Header                                                          */
/*****************************************************************************/
#include "devioctl.h"

#ifndef __USBDI_H__
typedef struct _USB_DEVICE_DESCRIPTOR {
   UCHAR  bLength;
   UCHAR  bDescriptorType;
   USHORT bcdUSB;
   UCHAR  bDeviceClass;
   UCHAR  bDeviceSubClass;
   UCHAR  bDeviceProtocol;
   UCHAR  bMaxPacketSize0;
   USHORT idVendor;
   USHORT idProduct;
   USHORT bcdDevice;
   UCHAR  iManufacturer;
   UCHAR  iProduct;
   UCHAR  iSerialNumber;
   UCHAR  bNumConfigurations;
} USB_DEVICE_DESCRIPTOR, *PUSB_DEVICE_DESCRIPTOR;
#endif

#define  SYSFILE      "\\\\.\\USBSCAN%d"
#define MAX_USB_DEVICES  15

typedef struct _IO_BLOCK {
  unsigned    uOffset;
  unsigned    uLength;
  PBYTE       pbyData;
} IO_BLOCK, * PIO_BLOCK;

typedef struct _PIPE_LIST{
  OUT   unsigned    EventChannelSize;
  OUT   unsigned    uReadDataAlignment;
  OUT   unsigned    uWriteDataAlignment;
} CHANNEL_INFO, * PCHANNEL_INFO;


typedef enum {
```

```
  EVENT_PIPE,
  READ_DATA_PIPE,
  WRITE_DATA_PIPE,
  ALL
} PIPE_TYPE;

//---------------------------------------------------- definition
#define FILE_DEVICE_USB_SCAN  0x8000
#define IOCTL_INDEX           0x800

#define IOCTL_GET_VERSION \
  CTL_CODE(FILE_DEVICE_USB_SCAN,IOCTL_INDEX,METHOD_BUFFERED,FILE_ANY_ACCESS)

#define IOCTL_ABORT_PIPE \
  CTL_CODE(FILE_DEVICE_USB_SCAN,IOCTL_INDEX+1,METHOD_BUFFERED,FILE_ANY_ACCESS)

#define IOCTL_WAIT_ON_DEVICE_EVENT\
  CTL_CODE(FILE_DEVICE_USB_SCAN,IOCTL_INDEX+2,METHOD_BUFFERED,FILE_ANY_ACCESS)

#define IOCTL_READ_REGISTERS \
  CTL_CODE(FILE_DEVICE_USB_SCAN,IOCTL_INDEX+3,METHOD_BUFFERED,FILE_ANY_ACCESS)

#define IOCTL_WRITE_REGISTERS \
  CTL_CODE(FILE_DEVICE_USB_SCAN,IOCTL_INDEX+4,METHOD_BUFFERED,FILE_ANY_ACCESS)

#define IOCTL_GET_CHANNEL_ALIGN_RQST \
  CTL_CODE(FILE_DEVICE_USB_SCAN,IOCTL_INDEX+5,METHOD_BUFFERED,FILE_ANY_ACCESS)

BOOL FAR PASCAL CloseUSB(VOID);
BOOL FAR PASCAL FindScanner(WORD ScannerProd);
BOOL UsbCmdRead(BYTE cmd, WORD n, PBYTE pData);
BOOL UsbCmdWrite(BYTE cmd, WORD n, PBYTE pData);
BOOL UsbDataRead(DWORD n, PVOID pData);

/*****************************************************************************
 *  slusb.cpp:  usbscan.sys interface
 *  Copyright:  (c)1997 ScanLogic Corporation
 *              4 Preston Court, Bedford, MA 01739
 *  Revision:   1.0 July/14/96       SNguyen
*****************************************************************************/
#include "stdafx.h"
#pragma hdrstop

#include <windows.h>
#include "slusb.h"

static HANDLE hDev=0;




/*****************************************************************************
 * UsbCmdRead
 * Description :    UsbCmdRead
```

```
 * Parameters :    hDev
 *                                        BYTE    cmd
 *                                        WORD    nb of data to read
 *                                        PBYTE   ptr that receive data
 * Return     :    BOOL    TRUE means successful, else FAIL
 ***************************************************************************/
BOOL UsbCmdRead(BYTE cmd, WORD wNbOfReg, PBYTE pbyData)
{  DWORD      cbRet;
   IO_BLOCK   IoBlock;

   IoBlock.uOffset = cmd;
   IoBlock.uLength = (WORD)wNbOfReg;
   IoBlock.pbyData = pbyData;
   return DeviceIoControl(hDev,
                          (DWORD) IOCTL_READ_REGISTERS,
                          (PVOID)&IoBlock,
                          (DWORD)sizeof(IO_BLOCK),
                          (PVOID)pbyData,
                          (DWORD)wNbOfReg,
                          &cbRet,
                          NULL);
}

/***************************************************************************/
BOOL FAR PASCAL CloseUSB(VOID)
{
   if(hDev)
   {  CloseHandle(hDev);
      hDev=0;
      return TRUE;
   }
   return FALSE;
}
```

```
/***************************************************************************
 * UsbCmdWrite
 * Description :    Write cmd block to Scanner
 * Parameters :    hDev
```

```
 *                                          BYTE   cmd
 *                                          WORD  nb of data to write
 *                                          PBYTE  ptr that write to scanner
 * Return     :      BOOL    TRUE means successful, else FAIL
 ***************************************************************************/
BOOL UsbCmdWrite(BYTE cmd, WORD wNbOfReg, PBYTE pbyData)
{
  DWORD     cbRet;
  IO_BLOCK   IoBlock;
  IoBlock.uOffset = cmd;
  IoBlock.uLength = (WORD)wNbOfReg;
  IoBlock.pbyData = pbyData;
  return DeviceIoControl(hDev,
                                      (DWORD) IOCTL_WRITE_REGISTERS,
                                      &IoBlock,
                                      sizeof(IO_BLOCK),
                                      NULL,
                                      0,
                                      &cbRet,
                                      NULL);
}

/***************************************************************************/
BOOL FAR PASCAL FindScanner(WORD ScannerProd)
{  int i;
  DWORD cbRet;
  USB_DEVICE_DESCRIPTOR dev_desc;
  char cFileName[30];

  if(!hDev)
  {   for(i=0;i<MAX_USB_DEVICES;i++)
     {
        wsprintf(cFileName,SYSFILE,i);
        if ((hDev=CreateFile(cFileName,GENERIC_WRITE|GENERIC_READ,0,NULL,
                  OPEN_EXISTING,0,NULL )) !=  INVALID_HANDLE_VALUE)
        {  if((DeviceIoControl(hDev,(DWORD)IOCTL_GET_VERSION,NULL,0,(PVOID)&dev_desc,
                    sizeof(USB_DEVICE_DESCRIPTOR),&cbRet,NULL)))
          {  if (dev_desc.idProduct == ScannerProd) return TRUE;
                }
          CloseHandle(hDev);
        }
     } // for
     hDev=0;
     return FALSE;
  }
  return TRUE;
}
```

```
/*****************************************************************************/
BOOL UsbDataRead(DWORD cData,PVOID pData)
{   DWORD       dwRet;

   ReadFile(hDev, pData, cData, &dwRet, NULL);
   if (dwRet<cData) return FALSE;
   return TRUE;
}

/*****************************************************************************
 *  usbtestDlg.cpp : implementation file
 *  Copyright:  (c)1997 ScanLogic Corporation
 *              4 Preston Court, Bedford, MA 01739
 *  Revision:   1.0 July/14/96        SNguyen
 *****************************************************************************/

const WORD  SCANNER_ID = 0x300;
const WORD  DEF_LINENUMBER= 100;
const WORD  DEF_LINESIZE = (4096*8);

#define CMD_WR_LEN              (BYTE)8
#define CMD_RD_LEN              (BYTE)2
#define LAMP_OFF                (BYTE)0x00
#define GET_FW_VERSION          (BYTE)0x01
#define GET_DPI                 (BYTE)0x02
#define SET_IMAGE_SIZE          (BYTE)0x03
#define LOOPBACK_TEST           (BYTE)0x04
#define LAMP_ON                 (BYTE)0x05

static BYTE OutBuf[CMD_WR_LEN], OutCmd, InCmd, InBuf[CMD_RD_LEN];

//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////
void CUsbtestDlg::OnGetdpiButton()
{
            // TODO: Add your control notification handler code here
            CWnd* pGetDpi = GetDlgItem(IDC_GETDPI);
            if(IsScannerFound)
   { CString ss;
              if (UsbCmdRead(GET_DPI, CMD_RD_LEN, InBuf))
    {   ss.Format("0x%x", InBuf[0]);
                  pGetDpi->SetWindowText(ss);
    }
            }
            else pGetDpi->SetWindowText("N/A");
}




//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////
```

```
void CUsbtestDlg::OnGetfirmwareButton()
{
        // TODO: Add your control notification handler code here
        CWnd* pGet = GetDlgItem(IDC_GETFIRMWARE);
        if(IsScannerFound)
   { CString ss;

    if(UsbCmdRead(GET_FW_VERSION, CMD_RD_LEN, InBuf))
    {   ss.Format("0x%x", InBuf[0]);
            pGet->SetWindowText(ss);
    }
        }
        else pGet->SetWindowText("N/A");
}

//////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////
void CUsbtestDlg::OnTestScannerButton()
{
        // TODO: Add your control notification handler code here
        CWnd* pGet = GetDlgItem(IDC_TESTSCANNER);
        if(IsScannerFound)
   { CString ss;

    if(UsbCmdRead(LOOPBACK_TEST, CMD_RD_LEN, InBuf))
    {   ss.Format("0x%x", InBuf[0]);
            pGet->SetWindowText(ss);
    }
        }
        else pGet->SetWindowText("N/A");
}

//////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////
void CUsbtestDlg::OnSaveFileButton()
{
        CWnd* pGet = GetDlgItem(IDC_SAVE_FILE);
        CString ss;

        SaveFile = !SaveFile;
        if (SaveFile) ss.Format("Yes"); else ss.Format("No ");
        pGet->SetWindowText(ss);

}
```

```
//////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////
void CUsbtestDlg::OnUsbWriteButton()
{
```

```
        CWnd* pGet = GetDlgItem(IDC_USB_WRITE);
   CString ss;
        DWORD t[CMD_WR_LEN+1], i;

        pGet->GetWindowText(ss);
        sscanf(ss,"%02x %02x %02x %02x %02x", &t[0], &t[1], &t[2], &t[3], &t[4]);
   OutCmd = (BYTE)t[0];
        for (i=0; i<CMD_WR_LEN; i++) OutBuf[i]= (BYTE) t[i+1];

        if(IsScannerFound)
   {
    if(!UsbCmdWrite(OutCmd, 32, OutBuf))
    {   pGet->SetWindowText("UsbWrite Fail");
    }
        }
        else pGet->SetWindowText("00 00 00 00 00");
}


//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////
void CUsbtestDlg::OnUsbReadButton()
{
        CWnd* pGet = GetDlgItem(IDC_USB_READ);
        CString ss;

        pGet = GetDlgItem(IDC_USB_READ);
        InCmd = atoi(ss);

        if(IsScannerFound)
   {
    if(!UsbCmdRead(InCmd, CMD_RD_LEN, InBuf))
    {   pGet->SetWindowText("UsbRead Fail");
    }
    ss.Format("%02x %02x %02x", InCmd,  InBuf[0], InBuf[1]);
          pGet->SetWindowText(ss);
        }
        else pGet->SetWindowText("00");
}
```

```
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
void CUsbtestDlg::OnLampButton()
{
        CWnd* pGetDpi = GetDlgItem(IDC_LAMP);
        if(IsScannerFound)
   { CString ss;

         if (LampMode)
    {   if(UsbCmdRead(LAMP_ON, CMD_RD_LEN, InBuf))
       {   ss.Format("ON ");
              pGetDpi->SetWindowText(ss);
       }
         } else
         {   if(UsbCmdRead(LAMP_OFF, CMD_RD_LEN, InBuf))
       {   ss.Format("OFF");
              pGetDpi->SetWindowText(ss);
       }
         }
         LampMode = !LampMode;
        }
        else pGetDpi->SetWindowText("N/A");
}

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
void CUsbtestDlg::OnGetimageButton()
{
        // TODO: Add your control notification handler code here
        CWnd* pGetDpi = GetDlgItem(IDC_GETIMAGE);
        CString ss;
        CEdit* pLineNumber= (CEdit*)GetDlgItem(IDC_LINENUMBER);
        CEdit* pImageSize = (CEdit*)GetDlgItem(IDC_IMAGESIZE);
        pLineNumber->GetWindowText(ss);
        short linecnt  = atoi(ss);
        pImageSize->GetWindowText(ss);
        short imagesize = atoi(ss);


        // make sure the imagesize if multiple of 4, for creating bmpfile
        realimagesize = imagesize%4 == 0 ? imagesize:(imagesize/4-1)*4;

        if(imagesize == 0)
   {  AfxMessageBox("Please enter line number and image size!");
          return;
        }

        UserAbort = FALSE;
        if(IsScannerFound)
   {
         OutBuf[0] = imagesize>>8;          // High
    OutBuf[1] = imagesize&0xff;    // low
         OutBuf[2] = linecnt>>8;      // High
    OutBuf[3] = linecnt&0xff;          // low
    if(!UsbCmdWrite(SET_IMAGE_SIZE, CMD_WR_LEN, OutBuf))
    {  AfxMessageBox("UsbCmdWrite ImageSize fail!");
      return;
    }
```

```
    EnableAllControl(FALSE);
    GetDlgItem(IDC_TIMER)->SetWindowText(" ");
    DWORD starttime = GetTickCount();

    // if bmpfile not exist, create the file
    // or open the file for write
    m_BmpFilePath = GetBmpFilePath();
    if(::GetFileAttributes(m_BmpFilePath) != 0xffffffff)::DeleteFile(m_BmpFilePath);

    if(!BmpFile.Open(m_BmpFilePath, CFile::modeCreate|CFile::modeWrite|CFile::typeBinary))
    {    AfxMessageBox("create bmp file fails!");
         return;
    }

    GetDlgItem(IDC_IMAGEFILE)->SetWindowText(m_BmpFilePath);
    short ImageLineGot = ShowImageLine(linecnt, imagesize);

    DWORD endtime = GetTickCount();
    endtime -= starttime;
    CString ss;
    ss.Format("%d", endtime/1000);
    GetDlgItem(IDC_TIMER)->SetWindowText(ss);

    ::SetEndOfFile((HANDLE)BmpFile.m_hFile);
    BmpFile.Close();

    if(ImageLineGot)
    {   if(!MakeBmpFileHeader(ImageLineGot, realimagesize))
              BmpFile.Remove(m_BmpFilePath);
    }
    else BmpFile.Remove(m_BmpFilePath);
    EnableAllControl(TRUE);
         }
         else pGetDpi->SetWindowText("N/A");
}
```

SECTION 10

SL11 - SAMPLE FIRMWARE, INTGRATION PROGRAMMING (SCANNER APPLICATION)

Other source code examples are available and will be provided with SL11 Development Kit.

```
//-------------------------------------------------------------------------
//  sl11.inc :  SL11 definition
//  Copyright:  (c)1997 ScanLogic Corporation
//              4 Preston Court, Bedford, MA 01739
//  Revision:   May/22/96       SNguyen       Create
//  SL11 Firmware_Integrataion Sample Program
//-------------------------------------------------------------------------

/*
 * SL11 Register memory map
 */
typedef unsigned char  byte;
typedef unsigned short word;
typedef unsigned char  idata ibyte;
typedef unsigned short idata iword;

typedef union {
   byte bSize[4];
   word wSize[2];
} idata tImgData;

/*
 * SL11 Local Processor Memory MAP
 */

#define  cUSB_ADDR_CS    (*(byte xdata *)0x7a80)
#define  cUSB_DATA_CS    (*(byte xdata *)0x7a80)
sbit P11_DMA     = 0x91;
sbit P15_A0      = 0x95;
sbit P16_Reset   = 0x96;

#define     CtrlReg      0x05
#define     IntStatus    0x0D
#define       EP0Bit       0x01       // IntStatus Bit definition
#define       EP1Bit       0x02
#define       EP2Bit       0x04
#define       EP3Bit       0x08
#define       DMADoneBit   0x10
#define       ResetBit     0x40
#define       ProgressBit  0x80


#define  USBAdd         0x07
#define  IntEna         0x06
#define  DMACntLow      0x35
```

```
#define  DMACntHigh     0x36
#define  IntMask0       0x51                    /* Reset|DMA|EP0                */
#define  HostMask       0x41                    /*  Host request command        */
#define  ReadMask       0xd1                    /* Read mask interrupt          */


#define  EP0Control     00
#define  EP0Address     01
#define  EP0XferLen     02
#define  EP0Status      03
#define  EP0Counter     04

#define  EP1AControl    0x10
#define  EP1AAddress    0x11
#define  EP1AXferLen    0x12
#define  EP1ACounter    0x14

#define  EP1BControl    0x18
#define  EP1BAddress    0x19
#define  EP1BXferLen    0x1a
#define  EP1BCounter    0x1c

#define  EP2AControl    0x20
#define  EP2AAddress    0x21
#define  EP2AXferLen    0x22
#define  EP2ACounter    0x24

#define  EP2BControl    0x28
#define  EP2BAddress    0x29
#define  EP2BXferLen    0x2a
#define  EP2BCounter    0x2c

#define  EP3AControl    0x30
#define  EP3AAddress    0x31
#define  EP3AXferLen    0x32
#define  EP3ACounter    0x34

#define  EP3BControl    0x38
#define  EP3BAddress    0x39
#define  EP3BXferLen    0x3a
#define  EP3BCounter    0x3c


#define  ConfBuf        0x40  /* SL11 address start for configuration        */
#define  ConfBufLen     64    /* length of config buffer ConfBuf             */

#define  CmdRcv         0x40  /* SL11 address start for Receive Data from host */
#define  CmdRcvLen      64    /* SL11 total length to receive data           */

#define  CmdSnd         0x40  /* SL11 address start for Sending Data to host  */
#define  CmdSndLen      64    /* SL11 total length to sending data to host   */

#define  DMAout0        0x80  /* SL11 DMA address for DATA0                  */
#define  DMAout1        0xc0  /* SL11 DMA address for DATA1                  */
#define  DMAXferLen     64    /* SL11 DMA xfer length                        */

#define  nEndPoints     1  /* EndPoint 0 = host config, EndPoint 3 = DATA */
#define  TotalLen       25
```

```
/*****************************************************************************
** Host interface command
*****************************************************************************/
#define  FW_VERSION     0x10

#define  GET_STATUS      0x00
#define  CLEAR_FEATURE   0x01
#define  SET_FEATURE     0x03
#define  SET_ADDRESS     0x05
#define  GET_DESCRIPTOR  0x06
#define  SET_DESCRIPTOR  0x07
#define  GET_CONFIG      0x08
#define  SET_CONFIG      0x09
#define  GET_INTERFACE   0x0a
#define  SET_INTERFACE   0x0b

#define  DEVICE          0x01
#define  CONFIGURATION   0x02
#define  STRING          0x03
#define  INTERFACE       0x04
#define  ENDPOINT        0x05

/*
 * Application Command Interfaces
 */

#define GET_FW_VERSION       0x00
#define USB_CMD_READ         0x01
#define INQUIRY              0x02
#define SET_IMAGE_SIZE       0x03
#define LOOPBACK_TEST        0x04
#define USB_CMD_WRITE        0x05

/*
 * Global Variables
 */

bit EP3DataBit;          // Endpoint 3 Data0 & Data1 Flag
bit OddDmaBit;           // Check DMA Odd or Even Transfer on 64 byte packet
bit RemDmaBit;           // Remainder Flag if DMA Transfer not divisible by 64
bit InDMA;               // Check if Current ASIC DMA is enabled
bit EP0DataBit;          // EndPoint 0 Data0 & Data1 Flag
bit IOTCLReadBit;        // Vendor Commmand Read/Write Flag

tImgData   ImgData;      // Image Window information
ibyte      bDeviceAddr;  // USB Device Address asigned by host
ibyte      bUsbIOTCLCmd; // Vendor Command send via host IOTCL call
iword      wUsbCmdLen;   // Vendor Command Length
ibyte      bUsbStatus;   // Command transfer Status checking

void SL11Write(byte addr, byte cData)
{
  P15_A0 = 0;
  cUSB_ADDR_CS = addr;             /* setup address */
  P15_A0 = 1;
  cUSB_DATA_CS = cData;  /* write data into adress */
}
```

```
byte SL11Read(byte addr)
{
  P15_A0 = 0;
        cUSB_ADDR_CS = addr;
  P15_A0 = 1;
        return cUSB_DATA_CS;
}

void SL11ReadBlock(byte SrcAddr, ibyte *DstAddr, byte len)
{
  P15_A0 = 0;
        cUSB_ADDR_CS = SrcAddr;
  P15_A0 = 1;
        while (len--)
                *DstAddr++ = cUSB_DATA_CS;
}

void SL11WriteBlock(const byte code *SrcAddr, byte DstAddr, byte len)
{
        register byte bTmp = len;
  P15_A0 = 0;
        cUSB_ADDR_CS= DstAddr;
  P15_A0 = 1;
        while (len--)
                cUSB_DATA_CS = *SrcAddr++;
        SetSendCmd(bTmp);       // send to the host
}

byte SL11MemTest(void )
{
  register byte bTmp;

        SL11Write(ConfBuf,0x5a);
        if (0x5a != SL11Read(ConfBuf))
                return 1;

        SL11Write(ConfBuf, 0xa5);
        if (0xa5 != SL11Read(ConfBuf))
                return 2;

  P15_A0 = 0;
  cUSB_ADDR_CS = ConfBuf;
  P15_A0 = 1;
  for (bTmp=ConfBuf; bTmp < 0xff; bTmp++)
    cUSB_DATA_CS = bTmp;

  P15_A0 = 0;
  for (bTmp = cUSB_ADDR_CS = ConfBuf, P15_A0=1; bTmp < 0xff; bTmp++)
    if (bTmp != cUSB_DATA_CS) return 3;

  for (bTmp = ConfBuf; bTmp < 0xff; bTmp++)
  {
    if (bTmp != SL11Read(bTmp)) return 4;
    SL11Write(bTmp,~bTmp);
    if (SL11Read(bTmp) != ~bTmp) return 5;
  }
  return 0;
```

```
}

void SetupSL11()
{
   ImgData.wSize[1] =
   bDeviceAddr      =
   EP3DataBit       =
   bUsbIOTCLCmd     =
   wUsbCmdLen       = 0;
   SL11Write(USBAdd,0);          // set address = 0
   SL11Write(EP0Counter,0);      // clear all endpoint counters
   SL11Write(EP0Address,ConfBuf);
   SL11Write(EP0XferLen,ConfBufLen);
   SL11Write(EP0Control,03);

   SL11Write(EP3AAddress,DMAout0);
   SL11Write(EP3BAddress,DMAout1);

   SL11Write(CtrlReg,3);         // enable SL11 DMA
   SL11Write(IntEna,IntMask0);   // set interupt mask
   SL11Write(IntStatus,0xff);    // Clear the interrupt all flags
}

void SL11EnableDma(void )
{
        if (EP3DataBit) {
                SL11Write(EP3AControl, 0x4e);
                SL11Write(EP3BControl, 0x6);
        } else {
                SL11Write(EP3AControl, 0xe);
                SL11Write(EP3BControl, 0x46);
        }
        SL11Write(EP3AXferLen, DMAXferLen);
        SL11Write(EP3BXferLen, DMAXferLen);
        SL11Write(DMACntLow,   ImgData.bSize[1]);
        SL11Write(DMACntHigh,  ImgData.bSize[0]);
}
```

```
/*
 * See Chapter 9 USB Specification version 1.0 for more information
 */
bit SwitchBuf;
ibyte count;

const byte code SL_DEV[]  = {0x12, DEVICE, 0,1,0xff,3,0,0x40,0xce,4,0,3,0,0,0,0,1};
const byte code SL_CONF[] = {9,CONFIGURATION,TotalLen,0,1,1,0,0x40,0};
const byte code SL_INTF[] = {9,INTERFACE,0,0,nEndPoints,0,0,0,0};
const byte code SL_EP3[]  = {7,ENDPOINT,0x83,2,DMAXferLen,0,0};
```

```
void SetReceiveCmd( void )
{
   SL11Write(EP0XferLen, ConfBufLen);
   SL11Write(EP0Control, 0x03);
}

void SetSendCmd( byte length )
{
   SL11Write(EP0XferLen, length);
   SL11Write(EP0Control, 0x47);
}


byte Min(byte a, byte b) { if (a>b) return b; else return a; }

void SL11Int() interrupt 0
{
            byte status,len;

  if (InDMA) AsicDisableDma();

  status = SL11Read(IntStatus);
  if ( (status&ReadMask) == 0)
     return;   // No Interrupt should check other share interrupt

  if (status & ResetBit) // detect USB Reset (i.e Cable remove/insert)
  {   SetupSL11();
     return;
  }

  if (status & EP0Bit)   // interrupt from End Point 0
  {
     SL11Write(IntStatus, EP0Bit);           // clear EPO interrupt

                        if (bUsbIOTCLCmd)
     {
        ProcessIOTCLCmd();
        return;
     }

     len = SL11Read(EP0XferLen) - SL11Read(EP0Counter);

     // Check len <8 and Last packet is a write packet
     if (len<8 || (SL11Read(EP0Control)&4))
     {   // Set Ready to receive cmd from host
       if (bDeviceAddr)    // Host assign new device address
       {
          SL11Write(USBAdd, bDeviceAddr);
          bDeviceAddr=0;
       }
       SetReceiveCmd();
       return;
     }

     len = SL11Read(ConfBuf+6);       // chapter 9 only use 1 byte length
     SL11Write(EP0XferLen, len);      // New host xfer length
     status = SL11Read(ConfBuf);
```

```
if ( (status&0x40)==0 )          //Check BmRequestType byte
{                 switch(SL11Read(ConfBuf+1))  //this is Chapter 9 command
   {   case GET_STATUS:
         SL11Write(ConfBuf, 0x00);
         SL11Write(ConfBuf+1, 0x00);
                        status = (SL11Read(ConfBuf+4) & 0x0f) << 4;
         if (SL11Read(status) & 0x20)
            SL11Write(ConfBuf, 0x01);
         SetSendCmd(2);
         return;

      case CLEAR_FEATURE:
         if (status != 2) break;

         status = (SL11Read(ConfBuf+4) & 0x0f) << 4;
         len = SL11Read(status) & 0xdf;    // Clear Stall for EPA
         SL11Write(status, len);

         status += 8;   // Select End Point B
         len = SL11Read(status) | 0xdf;    // Clear Stall for EPB
         SL11Write(status, len);
         break;

      case SET_FEATURE:
         if (status != 2) break;
         status = (SL11Read(ConfBuf+4) & 0x0f) << 4;
         len = SL11Read(status) | 0x20;    // Set Stall for EPA
         SL11Write(status, len);

         status += 8;   // Select End Point B
         len = SL11Read(status) | 0x20;    // Set Stall for EPB
         SL11Write(status, len);
         break;

      case SET_ADDRESS:  // host asign new device address
         bDeviceAddr = SL11Read(ConfBuf+2);
         break;

      case GET_DESCRIPTOR:
         switch (SL11Read(ConfBuf+3)) //Get the Descriptor type
         {   case DEVICE:
               SL11WriteBlock(SL_DEV, ConfBuf, min(len, sizeof(SL_DEV)));
               return;

             case CONFIGURATION:
               SL11WriteBlock(SL_CONF, ConfBuf, min(len, TotalLen));
               return;

             case INTERFACE:
               SL11WriteBlock(SL_INTF, ConfBuf, min(len, sizeof(SL_INTF)));
               return;

             case ENDPOINT:
               SL11WriteBlock(SL_EP3, ConfBuf, min(len, sizeof(SL_EP3)));
               return;
         }
         break;
      case GET_CONFIG:
```

```
            SL11Write(ConfBuf,0x01);
            SetSendCmd(1);
            return;

          case GET_INTERFACE:
            SL11Write(ConfBuf,0x00);
            SetSendCmd(1);
            return;

          // case SET_CONFIG, SET_INTERFACE, SET_DESCRIPTOR:

        }
        SetSendCmd(0);
    }
    else  // this is Vendor Specific Commands
    {
      IOTCLReadBit = status&0x80;    // get bit from bmRequestType
      EP0DataBit = SwitchBuf = 0;
      bUsbIOTCLCmd = SL11Read(ConfBuf+2);
      wUsbCmdLen = (word)(SL11Read(ConfBuf+7)*256 + len);
                              count = 0;
      if (!IOTCLReadBit)
      {
        if (wUsbCmdLen>=64)
          SetReceiveCmd();  // XferLength=64 and Set Arm
        else
          SL11Write(EP0Control,0x03);
        return;             // Next interrupt will process Write Cmd
      }

      switch(bUsbIOTCLCmd)
      {   case GET_FW_VERSION:  // Get FirmWare Version
            SL11Write(ConfBuf,FW_VERSION);
            break;

          case INQUIRY:         // Get DPI
            SL11Write(ConfBuf, bUsbStatus);
            break;

          case LOOPBACK_TEST:
            SL11Write(ConfBuf,bUsbIOTCLCmd);
            break;

      }
      if (wUsbCmdLen < 64)
      { // EP0XferLen was already written above
        SL11Write(EP0Control, 0x47);
        bUsbIOTCLCmd = 0;  // done transfer
      }
    }
    return;
  } // DMADone Handle here
        else if (status & DMADoneBit)
  {   SL11Write(IntStatus, DMADoneBit);

    if (!ImgData.wSize[1]--) return;

    AsicDisableDma();     //it is OK to read the SL11 IO now
```

```
      //Wait for Host empty the SL11 buffer
      while (SL11Read(IntStatus) & ProgressBit);

                        len = ImgData.bSize[1] & 0x3f;

      if (OddDmaBit) EP3DataBit = ~EP3DataBit;

      if (RemDmaBit)
      {   if (OddDmaBit)
          {
             SL11Write(EP3AXferLen, len);
             SL11Write(EP3AControl, SL11Read(EP3AControl)|0x01);
             while (SL11Read(EP3AControl)&1);
          }
          else
          {
             SL11Write(EP3BXferLen, len);
             SL11Write(EP3BControl, SL11Read(EP3BControl)|0x01);
             while (SL11Read(EP3BControl)&1);
          }
      }
      if (ImgData.wSize[1])
      {       AsicClrRead();
          SL11EnableDma();
          AsicEnableDma();
      }
   }
}

//*****************************************************************************
// Process Vendor IOTCL from host

void ProcessIOTCLCmd( void )
{
   count++;

   EP0DataBit = ~EP0DataBit;
   if (IOTCLReadBit)
   {   switch(bUsbIOTCLCmd)
       {   case USB_CMD_READ:
           break;

       }
       return;

   }
   if (!SwitchBuf)
   {   SetSendCmd(0);   // toggle to Data0
       SwitchBuf=1;
       return;
   }
   switch(bUsbIOTCLCmd)
   {
       case SET_IMAGE_SIZE:
           SL11ReadBlock(ConfBuf, (byte*)&ImgData, 4);
           RemDmaBit = OddDmaBit = 0;
                                   if (ImgData.bSize[1] & 0x3f)
           {
```

```
            RemDmaBit = 1;
            if (!(ImgData.bSize[1] & 0x40))
                OddDmaBit = 1;
        }
        else
        {
            if (ImgData.bSize[1] & 0x40)
                OddDmaBit = 1;
        }
        if (ImgData.wSize[1])
        {
          AsicClrRead();
          SL11EnableDma();
          AsicEnableDma();
        }
        break;
  }

  if (wUsbCmdLen<64)
  {
    SetReceiveCmd();
    bUsbIOTCLCmd=0;
  }
  else
  {         bUsbStatus = 0;
    SL11ReadBlock(ConfBuf, (byte*)&ImgData, 4);
    if (wUsbCmdLen > 256)
    if (ImgData.bSize[0] != 0x66) bUsbStatus = 0xaa;
    // Read and check block here
            wUsbCmdLen -= 64;
    if (wUsbCmdLen < 64)
      SL11Write(EP0XferLen,wUsbCmdLen);
    else
      SL11Write(EP0XferLen,64);

    if (EP0DataBit)
      SL11Write(EP0Control,0x43);
    else
      SL11Write(EP0Control,0x03);

    SwitchBuf=0;

  }
}
```

SECTION 11

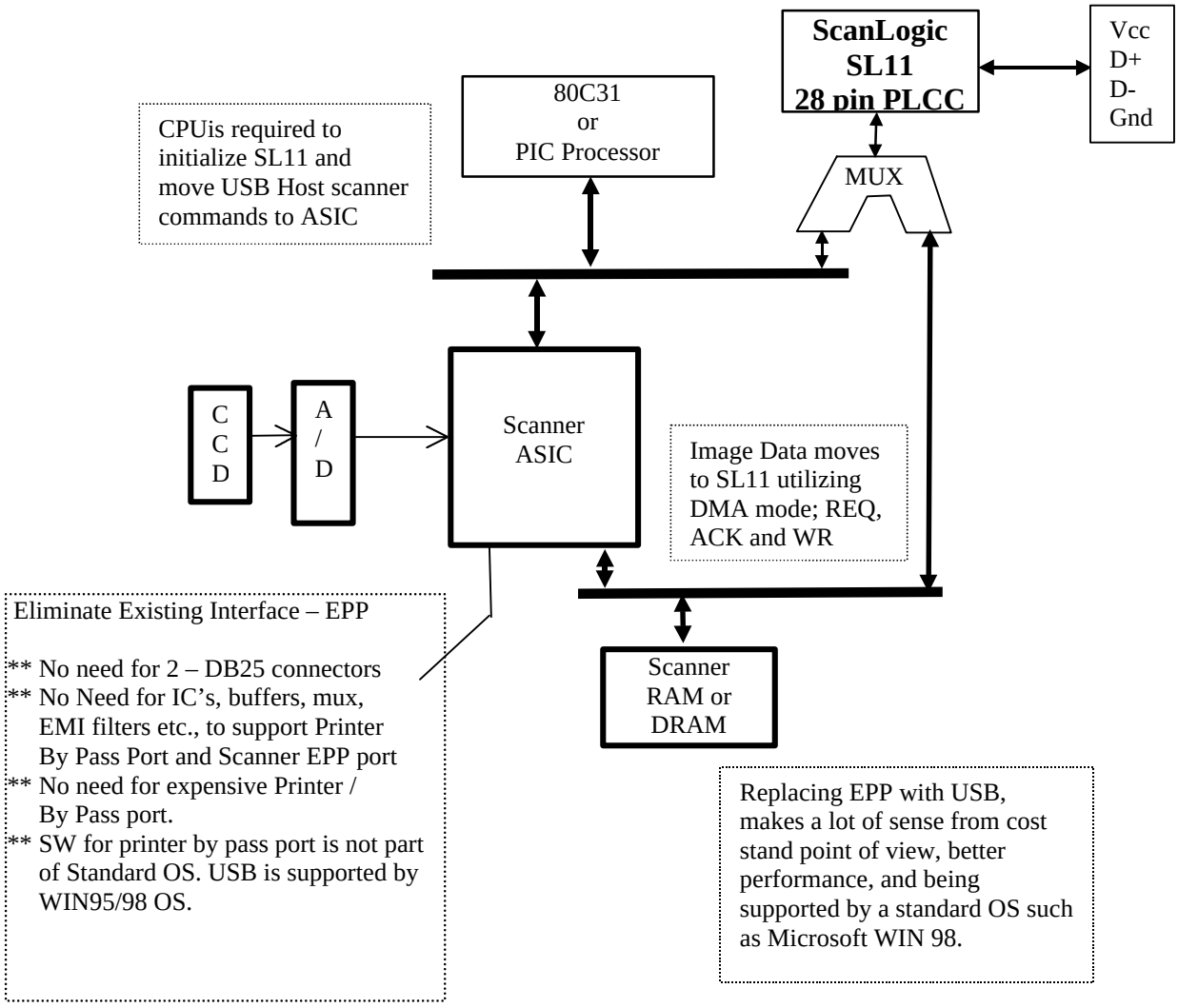
SL11 – DEVELOPMENT KIT (SL11DVK) ITIMIZED LIST:

5 Weeks to a working USB Peripheral

1. SL11/ISA demo card
2. SL11/ISA reference Design
3. SL11 Firmware examples source code.
4. USB Sample Demo source code. For USB host PC. Includes SL.INF file and instalation procedure.
5. USBSL.SYS Generic Mini-Port Driver for WIN95/98, source code option.
6. Application Notes
7. 2 – SL11 rev 1.1 sample chips
8. EMAIL support and training

ALL ABOVE ITEMS WILL BE PROVIDED UPON PURCHASING OF SL11 DEVELOPMENT KIT.

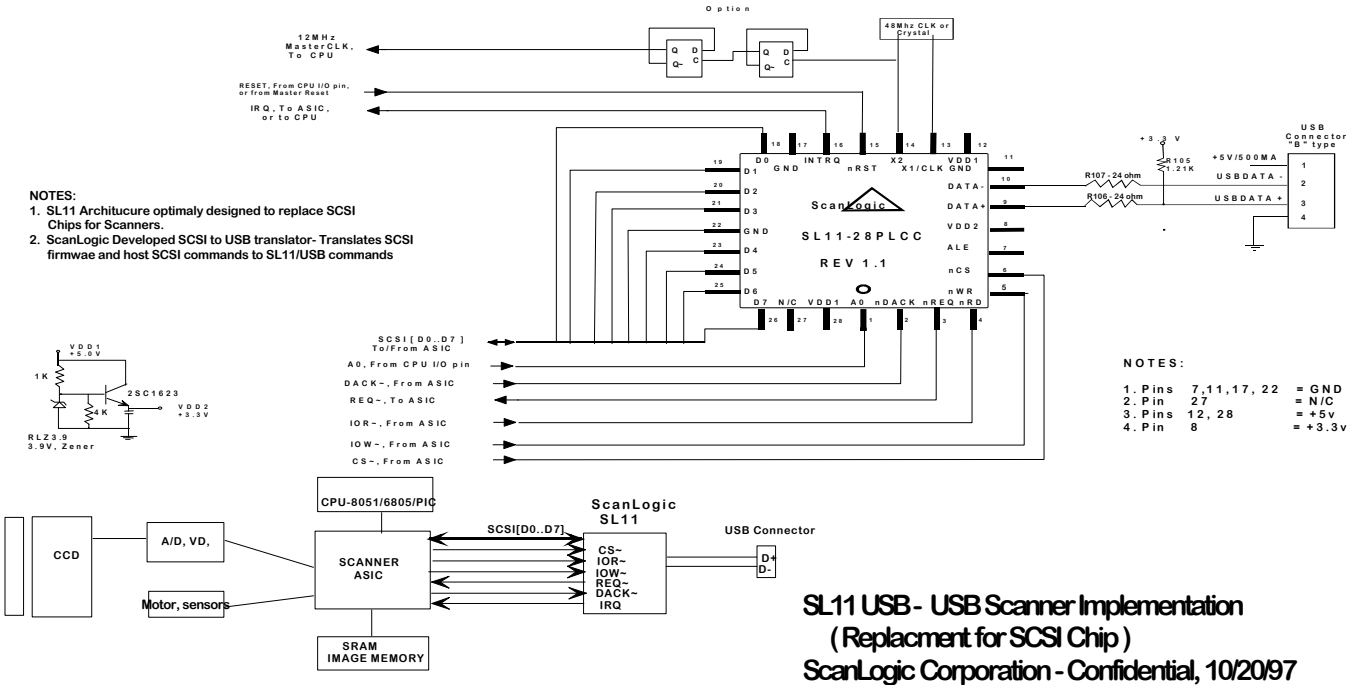
New Interface
USB Port



ScanLogic Corporation – SL11 USB Interface Chip
Scanner Application Block Diagram

Proprietary – 3/18/97

Implementation of SL11 - USB Interface to a Scanner. Design Example



EPP TO USB APPLICATION

This is an example of how to design an interface between EPP to the SL11/USB (see Figure 1). 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.